future internet

MDPI

Article

# Automatic Addition of Fault-Tolerance in Presence of Unchangeable Environment Actions †

**Mohammad Roohitavaf *,‡ and Sandeep Kulkarni *,‡**

Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA

* Correspondence: roohitavaf@gmail.com (M.R.); sandeep@cse.msu.edu (S.K.)

† This paper is an extended version of our paper: Roohitavaf, M.; Kulkarni, S. Stabilization and fault-tolerance in presence of unchangeable environment actions. In Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, Singapore 4–7 January 2016; p. 19.

‡ These authors contributed equally to this work.

Received: 15 April 2019; Accepted: 28 June 2019; Published: 4 July 2019


**Abstract:** We focus on the problem of adding fault-tolerance to an existing concurrent protocol in the presence of unchangeable environment actions. Such unchangeable actions occur in cases where a subset of components/processes cannot be modified since they represent third-party components or are constrained by physical laws. These actions differ from faults in that they are (1) simultaneously collaborative and disruptive, (2) essential for satisfying the specification and (3) possibly non-terminating. Hence, if these actions are modeled as faults while adding fault-tolerance, it causes existing model repair algorithms to declare failure to add fault-tolerance. We present a set of algorithms for adding stabilization and fault-tolerance for programs that run in the presence of environment actions. We prove the soundness, completeness and the complexity of our algorithms. We have implemented all of our algorithms using symbolic techniques in Java. The experimental results of our algorithms for various examples are also provided.

**Keywords:** stabilization; fault-tolerance; program synthesis; addition of fault-tolerance; model repair; cyber physical systems

## 1. Introduction

Model repair is the problem of revising an existing model/program so that it satisfies new properties while preserving existing properties. It is desirable in several contexts such as when an existing program needs to be deployed in a new setting or to repair bugs. Model repair for fault-tolerance enables one to separate the fault-tolerance and functionality so that the designer can focus on the functionality of the program and utilize automated techniques for adding fault-tolerance. It can also be used to add fault-tolerance against a newly discovered fault.

This paper focuses on performing such repair when some actions cannot be removed from the model. We refer to such transitions as unchangeable environment actions. There are several possible reasons for actions being unchangeable. Examples include scenarios where the system consists of several components –some of which are developed in-house and can be repaired and some of which are third-party and cannot be changed. They are also useful in systems such as Cyber-Physical Systems (CPS) where modifying physical components may be very expensive or even impossible.

The environment actions differ from fault actions considered in existing work such as [1]. Fault actions are assumed to be temporary in nature and all the previously proposed algorithms to add fault-tolerance in [1] work only with this important assumption that faults finally stop occurring. Unlike fault actions, environment actions can keep occurring. Environment actions also differ from adversary actions considered in [2] or in the context of security intrusions. In particular, the adversary

intends to cause harm to the system. By contrast, environment actions can be collaborative as well. In other words, environment actions are simultaneously collaborative and disruptive. The goal of this work is to identify whether it is possible for the program to be repaired so that it can utilize the assistance provided by the environment while overcoming its disruption. To give an intuition of the role of the environment and the difference between program, environment and fault actions, we present the following examples.

**Intuitive examples to illustrate the role of the environment.** The first intuitive example is motivated by a simple pressure cooker (see Figure 1). The environment (heat source) causes the pressure to increase. In the subsequent discussion, we analyze this pressure cooker when the heat source is always on. There are two mechanisms to decrease the pressure, a vent and an overpressure valve. For the sake of presentation, assume that pressure below 4 is normal. If the pressure increases to 4 or 5, the vent mechanism reduces the pressure by 1 in each step. However, the vent may fail (e.g., if something gets stuck in the vent pipe) and its pressure reduction mechanism becomes disabled. If the pressure reaches 6, the overpressure valve mechanism causes the valve to open resulting in an immediate drop in pressure to be less than 4. We denote the state where the pressure is $x$ by $s_x$ when the vent is working and by $fs_x$ when the vent has failed.

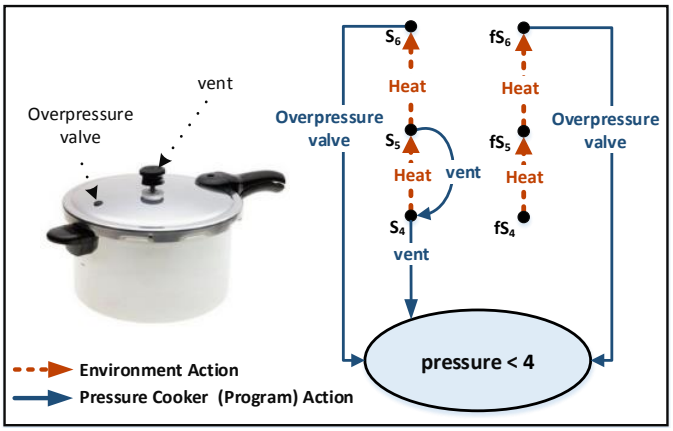


**Figure 1.** An intuitive example to illustrate the role of environment actions. We want the pressure to be always less than 4. The program captures the behavior of the pressure cooker including the vent and the overpressure valve. The environment captures the behaviors of the heat source. The environment can be disruptive for the venting mechanism. It is, however, essential for the overpressure valve to be activated. For the sake of readability, only transitions relevant to the discussion are shown in the figure.

Our goal in the subsequent discussion is to model the pressure cooker as a program and identify an approach for the role of the environment and its interaction with the program so that we can conclude this requirement: *starting* from *any* state identified above, the system reaches a state where the pressure is less than 4.

Next, we argue that how the role of the environment differs from the role of fault and program actions. In turn, this prevents us from using existing approaches such as Reference [1]. Specifically,

- Treating the environment as a fault does not work. Faults are assumed to be exceptional events in the system that are expected to stop after some time. By contrast, heat is an essential part of the system that is needed for the system to work. In addition, if we treat the environment as a fault, then none of the environment transitions including transitions from state $fs_4$ to $fs_5$ and from $fs_5$ to $fs_6$ are required to occur. If these actions do not occur, the overpressure valve is never activated. Hence, neither the valve nor the vent mechanism reduces the pressure to be less than 4.
- Treating the environment transitions similar to program transitions is also not acceptable. To illustrate this, consider the case where we want to make changes to the program in Figure 1. For instance, if the overpressure valve is removed, then this would correspond to removing the

transition from $s_6$ (respectively $fs_6$) to where the pressure is less than 4. Also, if we add another safety mechanism, it would correspond to adding new transitions. However, we cannot do the same with environment actions that capture the changes made by the heat source. For example, we cannot add new transitions (e.g., from $fs_4$ to $s_4$) to the environment and we cannot remove transitions (e.g., from $s_4$ to $s_5$). In other words, even if we make any changes to the model in Figure 1 by adding or removing safety mechanisms, the transitions marked as environment actions remain unchanged. We cannot introduce new environment transitions and we cannot remove existing environment transitions. This is what we mean by the environment being unchangeable.

- Treating the environment to be collaborative without some special fairness to the program does not work either. In particular, without some special fairness for the program, the system can cycle through states $s_4, s_5, s_4, s_5 \cdots$.
- Treating the environment to be simultaneously collaborative as well as adversarial where the program has some special fairness enables one to ensure that this program achieves its desired goals. In particular, we need the environment to be *collaborative*, that is, if it reaches a state where only environment actions can execute then one of them execute This is necessary to ensure that system can transition from state $fs_4$ to $fs_5$ and from $fs_5$ to $fs_6$ which is essential for recovery to a state where the pressure is less than 4. We also need the program to have special fairness to require that it executes *faster* than the environment so that it does not execute in a cycle through states $s_4, s_5, s_4, \cdots$ (we will precisely define the notion of faster in Section 2.1).

As another example, consider the shepherding problem. In this problem, a shepherd and his dog want to steer a sheep to a specific location (cf. Figure 2). They cannot carry the sheep; the only way to move the sheep is the movement of the sheep by itself. The sheep always tries to increase its distance from the shepherd and the dog. In this example, the program captures the behavior of the shepherd and the dog and the environment captures the behaviors of the sheep.

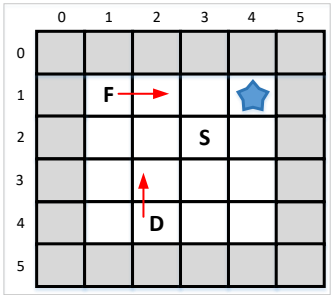


**Figure 2.** The farmer and the dog want to steer the sheep to the location marked by a star. The program captures the behavior of the farmer and the dog and the environment captures the behavior of the sheep. A good program uses the assistance of the sheep (i.e., the environment) while overcomes its disruption to steer it to the desired location.

Like the pressure cooker example, the environment is both assistive and disruptive. On one hand, the environment is assistive, because without the environment actions it is impossible to reach the desired state. On the other hand, the environment is disruptive, because the environment actions can make a (poorly designed) program go into an infinite loop thereby never reaching the desired state. A good program, however, uses the assistance of the sheep (i.e., the environment) while overcomes its disruption to steer the sheep to the desired location.

**Goal of the paper.** Based on the above examples, our goal in this paper is to evaluate how such simultaneously collaborative and adversarial environment can be used in adding stabilization and fault-tolerance to a given program.

A preliminary version of this work appeared in [3]. In addition to the results presented in [3], in this paper, we provide a sound and complete algorithm for adding masking fault-tolerance. We also introduce our JavaRepair package that includes implementation of our algorithms in Java using

symbolic techniques to support larger state spaces. We provide experimental results of using our proposed algorithms to solve different versions of the shepherding problem. To obtain these results, we have used our JavaRepair package. The main results of this work are as follows:

- We formalize definitions and problems statements for the addition of stabilization and fault-tolerance in presence of unchangeable environment actions.
- We present an algorithm for the addition of stabilization to an existing program. This algorithm is sound and complete, that is, the program found by it is guaranteed to be stabilizing and if it declares failure then it implies that adding stabilization to that program is impossible.
- We present a sound and complete algorithm for the addition of fail-safe fault-tolerance.
- We present a sound and complete algorithm for the addition of masking fault-tolerance.
- We show that the complexity of all algorithms presented in this paper is polynomial (in the state space of the program).
- We present our JavaRepair package that includes the implementation of the algorithms presented in this paper available at [4].
- We present experimental results of applying our algorithms for different examples.

**Organization of the paper.** This paper is organized as follows: in Section 2, we provide the definitions of a program, environment, specification, fault, fault-tolerance and stabilization. In Section 3 we define the problem of adding stabilization and propose an algorithm to solve that problem. In Section 4, as a case study, we illustrate how the proposed algorithm can be used for a smart grid controller. In Section 5, we define the problem of adding fault-tolerance and propose algorithms to add failsafe and masking fault-tolerance. In Section 6, we present our JavaRepir package and provide experimental results for several examples. In Section 7, we show how our proposed algorithms can be extended to solve related problems. In Section 8, we discuss related work. Finally, we make concluding remarks in Section 9.

## 2. Preliminaries

In this section, we define the notion of program, environment, specification, fault and fault-tolerance. The definition of the specification is based on the definition by Alpern and Schneider [5]. The definitions of fault and fault-tolerance are adapted from those by Arora and Gouda [6].

### *2.1. Program Design Model*

We define a program in terms of its states and transitions. Intuitively, the state space of a program represents the set of all possible states that a program can be in. On the other hand, transitions specify how the program moves from one state to another.

**Definition 1** (Program). *A program $p$ is a tuple $\langle S_p, \delta_p \rangle$ where $S_p$ is the state space of program $p$ and $\delta_p \subseteq S_p \times S_p$.*

In addition to program transitions, the state of a program may change due to the environment where the program is running. Instead of modeling the environment in terms of concepts such as variables that are written by program and variables that are written by the environment, we use a more general approach which models it as a set of transitions that is a subset of $S_p \times S_p$.

**Definition 2** (Environment). *An environment $\delta_e$ for program $p$, is defined as a subset of $S_p \times S_p$.*

Our algorithms assume that from any state in the state space of a program $p$ in an environment $\delta_e$, there is at least one transition in $\delta_p \cup \delta_e$. If there is no transition from state $s_0$ in $\delta_p \cup \delta_e$ of the given program, we add the self-loop transition $(s_0, s_0)$ to $\delta_p$. We note that this assumption is not restrictive.

Instead, it is made to simplify subsequent definitions, since we do not need to concern with *terminating computations* separately.

We refer to a subset of states of a program by a state predicate. Thus, we have

**Definition 3** (State Predicate)**.** *A state predicate of p is any subset of* $S_p$.

Note that we can represent a state predicate by either a subset of states or a predicate. Thus, there is a correspondence between a state predicate and subset of states where the state predicate is true. For example, state predicate *True* is a subset that includes all states of a program and *False* is an empty set. Similarly, the negation of a state predicate is the set of states where the given state predicate is false. We say a state predicate is closed in a set of transitions if no transition in the set leaves the state predicate.

**Definition 4** (Closure)**.** *A state predicate S is closed in a set of transitions* $\delta$ *iff* $(\forall(s_0, s_1) : (s_0, s_1) \in \delta : (s_0 \in S \Rightarrow s_1 \in S))$. *(We use* $Qx : D(x) : P(x)$ *to specify a formula where* $Qx$ *is a quantifier that is instantiated to be* $\forall x$ *or* $\exists x$, $D(x)$ *is a domain where x can come from and* $P(x)$ *is predicate over x. When there is no restriction on the domain, we use* $Q :: P$. *For example, '*$\exists x : x$ *is odd: x is prime' states that there exists an x in* $\{1, 3, 5, \cdots\}$ *that is a prime.)*

**Definition 5** (Projection)**.** *The projection of a set of transition* $\delta$ *on state predicate S, denoted as* $\delta|S$, *is set* $\{(s_0, s_1) : (s_0, s_1) \in \delta \wedge s_0, s_1 \in S\}\rangle$. *In other words,* $\delta|S$ *consists of transitions of* $\delta$ *that start at S and end in S.*

A computation of a program in an environment is a sequence of states that starts in an initial state and in each state executes either a program transition or an environment transition. Moreover, after an environment transition executes, the program is given a chance to execute in the next $k-1$ steps. However, whenever in a state no program transition is available, an environment transition can execute.

**Definition 6** ($p[]_k\delta_e$ *computation*)**.** *Let p be a program with state space* $S_p$ *and transitions* $\delta_p$. *Let* $\delta_e$ *be an environment for program p and k be an integer greater than 1. We say that a sequence* $\sigma = \langle s_0, s_1, s_2, ...\rangle$ *is a* $p[]_k\delta_e$ *computation iff*

- $\forall i : i \geq 0 : s_i \in S_p$, *and*
- $\forall i : i \geq 0 : (s_i, s_{i+1}) \in \delta_p \cup \delta_e$, *and*
- $\forall i : i \geq 0 : ((s_i, s_{i+1}) \in \delta_e) \Rightarrow$
  $(\forall l : i < l < i + k : (\exists s'_l :: (s_l, s'_l) \in \delta_p) \Rightarrow (s_l, s_{l+1}) \in \delta_p))$.

### 2.2. Specification

Following Alpern and Schneider [5], we let the specification of the program consist of a safety specification and a liveness specification. The safety specification identifies bad things that the program should not do. We define a safety specification as a set of (bad) transitions. Specifically,

**Definition 7** (Safety)**.** *A safety property is specified in terms of a set of bad transitions,* $\delta_b$, *that the program is not allowed to execute. Thus, a sequence* $\sigma = \langle s_0, s_1, \ldots\rangle$ *satisfies the safety property* $\delta_b$ *iff* $\forall j : j > 0 : (s_j, s_{j+1}) \notin \delta_b$.

Liveness, on the other hand, identifies good things that the program should do. We define a liveness property in terms of a set of leads-to properties. A leads-to property is of the form $L \rightsquigarrow T$ where $L$ and $T$ are two state predicates and the leads-to property requires that if the program reaches a state in $L$ it eventually reaches a state in $T$. Specifically,

**Definition 8** (Leads-to). *A leads-to property is specified in terms of $L \rightsquigarrow T$, where both $L$ and $T$ are state predicates. A sequence $\sigma = \langle s_0, s_1, \ldots \rangle$ satisfies the leads-to property iff $\forall j : s_j \in L : (\exists k : k \geq j : s_k \in T)$.*

Finally, a specification consists of a safety property and a liveness specification (in terms of a set of leads-to properties).

**Definition 9** (Specification). *A specification $spec$ , is a tuple $\langle Sf, Lv \rangle$, where $Sf$ is a safety property and $Lv$ is a set of leads-to properties. A sequence $\sigma$ satisfies $spec$ iff it satisfies $Sf$ and $Lv$.*

**Definition 10** (Satisfaction). *Program $p$ k-satisfies specification $spec$ from $S$ in environment $\delta_e$ iff the following conditions hold:*

- *$S$ is closed in $\delta_p \cup \delta_e$, and*
- *Every $p[]_k\delta_e$ computation that starts from a state in $S$ satisfies $spec$.*

We note that from the above definition, it follows that starting from a state in $S$, execution of either a program transition or an environment transition results in a state in $S$. Transitions that start from a state in $S$ and reach a state outside $S$ will be modeled as faults (cf. Definition 12). We define an invariant for a program as a subset of states such that if the program starts from that subset, it satisfies its required specification.

**Definition 11** (Invariant). *State predicate $S$ is an invariant of $p$ for specification $spec$ iff*

- *$S \neq \emptyset$, and*
- *$p$ k-satisfies $spec$ from $S$.*

### *2.3. Faults and Fault-Tolerance*

Like environment transitions, we define a class of faults for a program as a set of transitions that may change the state of the program:

**Definition 12** (Fault). *A class of faults $f$ for $p(= \langle S_p, \delta_p \rangle)$ is a subset of $S_p \times S_p$.*

Now, we extend Definition 6 to one that captures the computation of a program in presence of fault transitions:

**Definition 13** ($p[]_k\delta_e[]f$ *computation*). *Let $p$ be a program with state space $S_p$ and transitions $\delta_p$. Let $\delta_e$ be an environment for program $p$, $k$ be an integer greater than 1 and $f$ be the set of faults for program $p$. We say that a sequence $\sigma = \langle s_0, s_1, s_2, ... \rangle$ is a $p[]_k\delta_e[]f$ computation iff*

- *$\forall i : i \geq 0 : s_i \in S_p$, and*
- *$\forall i : i \geq 0 : (s_i, s_{i+1}) \in \delta_p \cup \delta_e \cup f$ , and*
- *$\forall i : i \geq 0 : (s_i, s_{i+1}) \in \delta_e \Rightarrow$*
  *$\forall l : i < l < i + k : (\exists s'_l :: (s_l, s'_l) \in \delta_p \Rightarrow (s_l, s_{l+1}) \in (\delta_p \cup f))$, and*
- *$\exists n : n \geq 0 : (\forall j : j > n : (s_{j-1}, s_j) \in (\delta_p \cup \delta_e))$.*

Note that in the above definition, we require that fault transitions finally stop occurring. This captures the transient nature of faults. On the other hand, environment transitions keep occurring forever. Fault transitions can perturb the program by arbitrarily changing its state. The definition of fault-span captures the boundary up to which the program could be perturbed by faults. Thus,

**Definition 14** (Fault-span). *The state predicate $T$ is a k-$f$-span of $p$ from $S$ in environment $\delta_e$ iff*

- *$S \Rightarrow T$, and*

- *for every* $p[]_k\delta_e[]f$ *computation* $\langle s_0, s_1, s_2, \ldots\rangle$, *where* $s_0 \in S$, $\forall i : s_i \in T$.

Next, we define the notion of fault-tolerance. We consider two different types of fault-tolerance namely failsafe and masking fault-tolerance. A failsafe fault-tolerance ensures that the safety property of the desired specification (cf. Definition 9) is not violated even if faults occur. In other words, we have

**Definition 15** (Failsafe fault-tolerance)**.** *Program* $p$ *is failsafe* $k$-$f$-*tolerant to specification spec* $(=\langle Sf, Lv\rangle)$ *from* $S$ *in environment* $\delta_e$ *iff the following two conditions hold:*

- $p$ *k-satisfies spec from* $S$ *in environment* $\delta_e$ *and*
- *any prefix of any* $p[]_k\delta_e[]f$ *computation that starts from* $S$ *satisfies* $Sf$.

Masking fault-tolerance is stronger than failsafe fault-tolerance, as in addition to satisfying the safety property, masking fault-tolerance requires recovery to the invariant. Specifically,

**Definition 16** (Masking fault-tolerance)**.** $p$ *is masking* $k$-$f$-*tolerant to specification spec from* $S$ *in environment* $\delta_e$ *iff the following two conditions hold:*

- $p$ *is failsafe* $k$-$f$-*tolerant to spec* $(=\langle Sf, Lv\rangle)$ *from* $S$ *in environment* $\delta_e$ *and*
- *there exists* $T$ *such that (1)* $T$ *is an* $k$-$f$-*span of* $p$ *from* $S$ *in environment* $\delta_e$ *and (2) for every* $p[]_k\delta_e[]f$ *computation* $\sigma(=\langle s_0, s_1, s_2, \ldots\rangle)$ *that starts from a state in* $S$, *for any* $i$ *such that* $s_i \in T$, *then there exists* $j \geq i$ *such that* $s_j \in S$.

The second constraint in Definition 16 simply means that in any computation that starts at $T$ must go to $S$.

We also define the notion of stabilizing programs. We extend the definition from References [7] and [8] for a program in the presence of environment actions.

**Definition 17** (Stabilization)**.** *Program* $p$ *is* $k$-*stabilizing for invariant* $S$ *in environment* $\delta_e$, *iff following conditions hold:*

- $S$ *is closed in* $\delta_p \cup \delta_e$, *and*
- *for any* $p[]_k\delta_e$ *computation* $\langle s_0, s_1, s_2, ...\rangle$ *there exists* $l$ *such that* $s_l \in S$.

## 3. Addition of Stabilization

In this section, we present our algorithm for adding stabilization to an existing program. In Section 3.1, we identify the problem statement and in Section 3.2, we present an algorithm to add stabilization. Finally, in Section 3.3, we provide proofs of soundness, completeness and the complexity results of the proposed algorithm.

### *3.1. Problem Definition*

The problem of adding of stabilization begins with a program $p = \langle S_p, \delta_p\rangle$, its invariant $S$ and an environment $\delta_e$. The goal is to change the set of program transitions so that starting from an arbitrary state, the program recovers to $S$. In addition, we do not want to change the behavior of the program inside its invariant. Thus, during the addition, we are not allowed to change the set of program transitions in the invariant (i.e., $\delta_p|S$). Also, we introduce parameter $\delta_r$ that identifies additional restrictions on program transitions. As an example, consider the case where a program cannot change the value of a sensor, that is, it can only read it. However, the environment can change the value of the sensor. In this case, transitions that change the value of the sensor are disallowed as program transitions but they are acceptable as environment transitions. Thus, the problem statement is as follows:

Given program $p$ with state space $S_p$ and transitions $\delta_p$, invariant $S$, environment $\delta_e$, set of program restriction $\delta_r$ and $k > 1$, identify $p'$ with state space $S_p$ such that:

**C1** $p'|S = p|S$
**C2** $p'$ is $k$-stabilizing for invariant $S$ in the environment $\delta_e$.
**C3** $\delta'_p \cap \delta_r = \emptyset$

### 3.2. Algorithm to Add Stabilization

The algorithm proposed here adds stabilization for the case where $k=2$. When $k=2$, environment transitions can execute immediately after any program transition. By contrast, for larger $k$, the environment transitions may have to wait until the program has executed $k-1$ transitions.

The procedure for adding stabilization is as shown in Algorithm 1. In this algorithm, $\delta'_p$ is the set of transitions of the final stabilizing program. Inside the invariant, the transitions must be equal to the original program (Constraint **C1**). Therefore, in the first line, we set $\delta'_p$ to $\delta_p|S$. Next, the algorithm expands set $R$ that includes states from which all computations reach a state in $S$. Initially, $R$ is set to $S$ at (Line 2).

State predicate $R_p$ is the set of states that can reach a state in $R$ using an unrestricted program transition, that is, a transition not in $\delta_r$. In each iteration, $R'_p$ is the set of new states that we add to $R_p$. In Line 9, we add program transitions from states in $R'_p$ to states in $R$ to $\delta'_p$.

**Algorithm 1** Addition of stabilization

**Input:** $S_p, \delta_p, \delta_e, S$, and $\delta_r$
**Output:** $\delta'_p$ or Not-Possible
1: $\delta'_p := (\delta_p|S)$;
2: $R = S$;
3: $R_p = \{\}$
4: **repeat**
5: $R' = R$;
6: $R'_p = \{s_0 | s_0 \notin (R \cup R_p) \wedge \exists s_1 : s_1 \in R : (s_0, s_1) \notin \delta_r\}$;
7: $R_p = R_p \cup R'_p$;
8: **for** *each* $s_0 \in R'_p$ **do**
9: $\delta'_p = \delta'_p \cup \{(s_0, s_1) | (s_0, s_1) \notin \delta_r \wedge s_1 \in R\}$;
10: **end for**
11: **for** *each* $s_0 \notin R : \nexists s_2 \in \neg(R \cup R_p) : (s_0, s_2) \in \delta_e \wedge (\exists s_1 : s_1 \in (R \cup R_p) : (s_0, s_1) \in \delta_e \vee s_0 \in R_p)$ **do**
12: $R = R \cup s_0$;
13: **end for**
14: **until** $(R' = R)$;
15: **if** $\exists s_0 \notin R$ **then**
16: **return** Not-Possible;
17: **else**
18: **return** $\delta'_p$;
19: **end if**

In the loop on Lines 11–13, we add more states to $R$. We add $s_0$ to $R$ (Line 12), whenever every computation starting from $s_0$ has a state in $S$. A state $s_0$ can be added to $R$ only when there is no environment transition starting from $s_0$ going to a state outside $R \cup R_p$. In addition to this condition, there must be at least one transition from $s_0$ that reaches $R$. The loop on Lines 4–14 terminates if no state is added to $R$ in an iteration. Upon termination of the loop, the algorithm declares failure to add stabilization if there exists a state outside $R$. Otherwise, it returns $\delta'_p$ as the set of transitions of the stabilizing program.

We use Figure 3 as an example to illustrate Algorithm 1. Figure 3 depicts the status of the state space in a hypothetical $i$th iteration of the loop on Lines 4–14. In this iteration, state **A** is added to $R$. This is due to the fact that (1) there is at least one transition from **A** (namely $(\mathbf{A}, \mathbf{F})$) that reaches $R$ and (2) there is no environment transition from **A** that reaches outside $R \cup R_p$. Likewise, state **C** is also added to $R$. State **B** is not added to $R$ due to environment transition $(\mathbf{B}, \mathbf{E})$. Likewise, state **D** is not added to $R$. State **E** is not added to $R$ since there is no transition from **E** to a state in $R$.

In the next, that is, $(i+1)$th iteration, **E** is added to $R$ due to transition $(\mathbf{E}, \mathbf{A})$ that goes to **A** that was added to $R$ in the $i$th iteration. Continuing this, **B** and **D** will be added in the $(i+2)$th iteration.

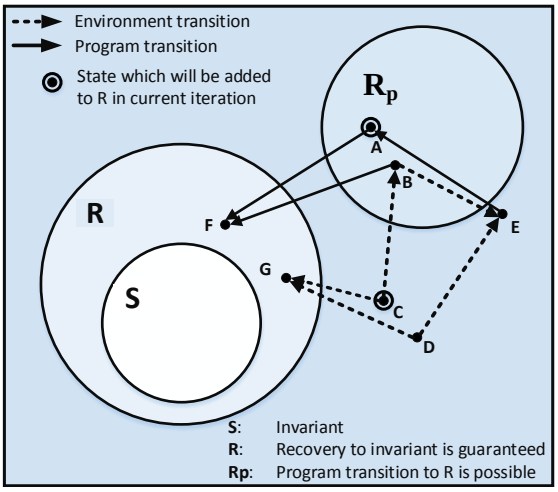


**Figure 3.** Illustration of how R expands in Algorithm 1.

### *3.3. Soundness, Completeness and Complexity Results*

In this Section, we show that Algorithm 1 is sound and complete and its time complexity is polynomial in the size of the state space.

**Soundness**: First, we show that Algorithm 1 is sound, that is, if it finds a solution, the solution satisfies the problem statement for adding stabilization provided in Section 3.1.

Based on the notion of fairness for program transitions, we introduce the notion of whether an environment transition can execute in a given computation prefix. An environment transition can execute in a computation prefix if an environment transition exists in the last state of the prefix and either (1) there are no program transitions starting from the last state of the prefix or (2) the last $k-1$ transitions of the prefix are program transitions.

**Definition 18** (Environment-enabled). *In any prefix of any $p[]_k\delta_e$ computation $\sigma = \langle s_0, s_1, \ldots, s_i \rangle$, $s_i$ is an environment-enabled state iff*

$$(\exists s :: (s_i, s) \in \delta_e) \wedge$$
$$\Big( (\nexists (s_i, s') :: (s_i, s') \in \delta_p) \vee$$
$$(\nexists j : j > i-k : (s_j, s_{j+1}) \in \delta_e) \Big).$$

The following lemma focuses on the recovery to $S$ from states in $R$:

**Lemma 1.** *Any $p'[]_k\delta_e$ computation that starts from a state in $R$, contains a state in $S$.*

**Proof.** We prove this by induction.
**Base case**: $R = S$. The statement is satisfied trivially.
**Induction hypothesis**: Theorem holds for the current set of $R$.
**Induction step**: We show when we add a state to $R$, the theorem still hold for $R$. A state $s_0$ is added to $R$ in two cases:

**Case 1** $(\nexists s_2 : s_2 \in \neg(R \cup R_p) : (s_0, s_2) \in \delta_e) \wedge (\exists s_1 : s_1 \in (R \cup R_p) : (s_0, s_1) \in \delta_e)$
Since there is no $s_2$ in $\neg(R \cup R_p)$ such that $(s_0, s_2) \in \delta_e$, for every $(s_0, s_1) \in \delta_e$, $s_1$ is in $R \cup R_p$. In addition, we know that there is at least one $s_1$ in $R \cup R_p$ such that $(s_0, s_1) \in \delta_e$. By the construction of $R_p$, we know that if $s_1$ is in $R_p$, there is a program transition from $s_1$ to a state in $R$. Since $(s_0, s_1) \in \delta_e$ and because of the fairness assumption, the program can occur and reach $R$. Thus, every computation starting from $s_0$ has a state in $R$. Since we do not change the set of transitions of states in $R$ of the previous iteration, the set of computations starting from any state in $R$ is unchanged. Thus, based on the induction hypothesis, every computation starting from $s_0$ has a state in $S$.

**Case 2** $\nexists s_2 : s_2 \in \neg(R \cup R_p) : (s_0, s_2) \in \delta_e \wedge s_0 \in R_p$
Since there is no $s_2$ in $\neg(R \cup R_p)$ such that $(s_0, s_2) \in \delta_e$, for every $(s_0, s_1) \in \delta_e$, $s_1$ is either in $R$ or $R_p$. In addition, we know that there is at least one state $s_3$ in $R$ such that $(s_0, s_3) \in \delta_p$. In any $p'[]_k\delta_e$ computation starting from $s_0$ if $(s_0, s_1) \in \delta'_p$ then $s_1 \in R$. If $(s_0, s_1) \in \delta_e$ then $s_1 \in R \cup R_p$. By construction of $R_p$, we know that if $s_1$ is in $R_p$, there is a program transition from $s_1$ to a state in $R$. Since $(s_0, s_1) \in \delta_e$ and because of the fairness assumption, program can reach $R$. Thus, every computation starting from $s_0$ has a state in $R$. Since we do not change the set of transitions of states in $R$ of the previous iteration, the set of computations starting from any state in $R$ is unchanged. Thus, based on the induction hypothesis, every computation starting from $s_0$ has a state in $S$. □

**Theorem 1.** *Algorithm 1 is sound.*

**Proof.** We need to show that the constraints of the problem definition are satisfied. At the beginning of the algorithm $\delta'_p = \delta_p|S$ and all other transitions added to $\delta'_p$ in the rest of the algorithm starts outside $S$. Thus, $p'|S = p|S$ that satisfies the first constraint. The second constraint is satisfied based on Lemma 1 and the fact that $R$ includes all states. The third constraint is satisfied, as we do not add any transitions in $\delta_r$ to the program. □

**Completeness**: Next, we focus on showing that Algorithm 1 is complete, that is, if there is a solution that satisfies the problem statement for adding stabilization, Algorithm 1 finds one. The proof of completeness is based on the analysis of states that are not in $R$ upon termination.

Any state that is not in $R$ at the end of Algorithm 1, either does not have any environment transition, or it has an environment transition that goes to $\neg(R \cup R_p)$. Thus, we note the following observation:

**Observation 1.** *For any $s_0$ such that $s_0 \notin R$ and $\exists s_1 :: (s_0, s_1) \in \delta_e$, we have $\exists s_2 : s_2 \in \neg(R \cup R_p) : (s_0, s_2) \in \delta_e$.*

Also, if a state is in $R_p$ but it is not in $R$, it has an environment transition to $\neg(R \cup R_p)$.

**Observation 2.** *For any $s_0 \in \neg R \cap R_p$, $\exists s_1 : s_1 \in \neg(R \cup R_p) : (s_0, s_1) \in \delta_e$.*

Now, for the rest of our discussion in this section, we assume that Algorithm 1 has returned failure. The following lemma focuses on the situation where a given revision $p''$ reaches a state marked as $\neg(R \cup R_p)$ by Algorithm 1.

**Lemma 2.** *Let $\delta''_p$ be any program such that $\delta''_p \cap \delta_r = \emptyset$. Let $s_j$ be any state in $\neg(R \cup R_p)$ at the end of Algorithm 1. Then, for every $p''[]_k\delta_e$ prefix $\alpha = \langle ..., s_{j-1}, s_j \rangle$, there exists suffix $\beta = \langle s_{j+1}, s_{j+2}, \ldots \rangle$, such that $\alpha\beta$ is a $p''[]_k\delta_e$ computation and one of two conditions below is correct:*

1. $s_{j+1} \in \neg(R \cup R_p)$
2. $s_{j+1} \in (R_p - R) \wedge s_{j+2} \in \neg(R \cup R_p)$

**Proof.** There are two cases for $s_j$:

**Case 1** If $s_j$ is environment-enabled in prefix $\alpha = \langle ..., s_{j-1}, s_j \rangle$
According to the Observation 1, since $s_j \in \neg(R \cup R_p)$, there exists $s'' \in \neg(R \cup R_p)$ such that $(s_j, s'') \in \delta_e$. We set $s_{j+1} = s''$.

**Case 2** If $s_j$ is not environment-enabled in $\alpha = \langle ..., s_{j-1}, s_j \rangle$
In this case $(s_j, s_{j+1}) \in \delta_p$ and as $s_j \in \neg(R \cup R_p)$, $s_{j+1} \in \neg R$ (otherwise, $s_j$ would be included in $R_p$, as we can reach state $s_{j+1} \in R$ with a transition which is not in $\delta_r$). There are two sub-cases for this case:

**Case 2.1** $s_{j+1} \in \neg R_p$
In this case $s_{j+1} \in \neg(R \cup R_p)$.

**Case 2.2** $s_{j+1} \in R_p$
As $s_{j+1} \in \neg R \cap R_p$, according to Observation 2, we have $\exists s_2 : s_2 \in \neg(R \cup R_p) : (s_{j+1}, s_2) \in \delta_e$. As $(s_j, s_{j+1}) \in \delta_p$, even with fairness $(s_{j+1}, s_2)$ can occur. Therefore we set $s_{j+2} = s_2$, that is, $s_{j+2} \in \neg(R \cup R_p)$. □

From this lemma, we conclude that for any given program $p''$, if a computation ever reaches a state in $\neg(R \cup R_p)$, there is a computation from that state that never reaches $R$. Specifically, we have the following corollary:

**Corollary 1.** *Let $\delta_p''$ be any program such that $\delta_p'' \cap \delta_r = \emptyset$. Let $s_j$ be any state in $\neg(R \cup R_p)$ at the end of Algorithm 1. Then for every $p''[]_k\delta_e$ prefix $\alpha = \langle ..., s_{j-1}, s_j \rangle$, there exists suffix $\beta = \langle s_{j+1}, s_{j+2}, \ldots \rangle$ (possibly $\langle\rangle$), such that $\alpha\beta$ is a $p''[]_k\delta_e$ computation and $\forall i : i \geq j : s_i \in \neg R$ (i.e., $\neg S$).*

**Theorem 2.** *Algorithm 1 is complete.*

**Proof.** Suppose program $p''$ solve the addition problem. Algorithm 1 returns Not-possible only when, at the end of loop on Lines 4–14 there exists a state $s_0$ such that $s_0 \notin R$. When $s_0 \notin R$, we have two cases as follows:

**Case 1** $\exists s_2 : s_2 \in \neg(R \cup R_p) : (s_0, s_2) \in \delta_e$
As there exists an environment action to state $s_2$ in $\neg(R \cup R_p)$, starting from $s_0$ there is a computation such that the next state after $s_0$ is in $\neg(R \cup R_p)$. Note that, when a computation starts from $s_0$, even with the fairness assumption $(s_0, s_2) \in \delta_e$ can occur. Based on Corollary 1, for every $\delta_p''$ such that $\delta_p'' \cap \delta_r = \emptyset$, starting from $s_0$, there is a computation such that all of its states are outside $R$ (i.e., outside $s$).

**Case 2** $\nexists s_1 : s_1 \in (R \cup R_p) : (s_0, s_1) \in \delta_e \wedge s_0 \notin R_p$
Based on Corollary 1, starting from $s_0 \in \neg(R \cup R_p)$, there is a computation such that every state is in $\neg R$. Therefore, for every $\delta_p''$ such that $\delta_p'' \cap \delta_r = \emptyset$, starting from $s_0$, there is a computation such that all of its states are outside $R$ (i.e., outside $s$).

Thus in both cases, $p''$ has a computation that never reaches the invariant (contradiction). □

**Time complexity**: Finally, regarding the time complexity of Algorithm 1 we have the following theorem:

**Theorem 3.** *Time complexity of Algorithm 1 is polynomial (in the size of state space of $p$).*

**Proof.** The proof follows from the fact that each statement in Algorithm 1 is executed in polynomial time and the number of iterations is also polynomial, as in each iteration at least one state is added to $R$. □

## 4. Case Study: Stabilization of a Smart Grid

In this section, we illustrate how Algorithm 1 can be used for adding stabilization to a controller program of a smart grid. We consider an abstract version of the smart grid described in [9] (see Figure 4). In this example, the system consists of a generator $G$ and two loads $Z_1$ and $Z_2$. There are three sensors in the system. Sensor G shows the power generated by the generator and sensors 1 and 2 show the demand of load $Z_1$ and $Z_2$, respectively. The goal is to ensure that proper load shading is used if the load is too high (respectively, generating capacity is too low).

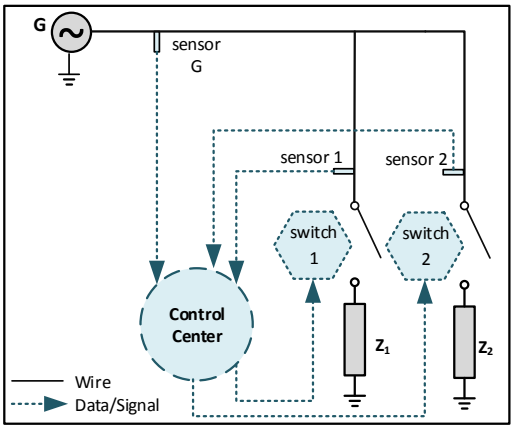


**Figure 4.** A single generator smart grid system [9].

The control center is shown by a dashed circle in Figure 4. It can read the values of the sensors and turn on/off switches connected to the loads. The program of the control center has to control switches in a manner that all the conditions below are satisfied:

1. Both switches must be turned on if the overall sensed load is less than or equal to the generation capacity.
2. If sensor values reveal that neither load can individually be served by G then both are shed.
3. If only one load can be served then the smaller load is shed assuming the larger load can be served by G.
4. If only one can be served and the larger load exceeds the generation capacity, the smaller load is served.

### *4.1. Program Model*

We model the program of the smart grid shown in Figure 4 by program $p$ which has five variables as follows:
$V_G$ : The value of sensor G.
$V_1$ : The value of sensor 1.
$V_2$ : The value of sensor 2.
$w_1$ : The status of switch 1.
$w_2$ : The status of switch 2.

The value of each sensor is an integer in the range $[0, max]$. The status of each switch is a Boolean. The invariant $S$ for this program includes all the states which are legitimate according to the conditions 1-4 mentioned above. Therefore, $S$ is the union of state predicates $I_1$ to $I_6$ as follows (We need to add $0 \leq V_1, V_2, V_g \leq max$ to all conditions. For brevity, we keep these implicit.):

$I_1 = (V_1 + V_1 \leq V_G) \wedge (w_1 \wedge w_2)$
$I_2 = V_1 \leq V_G \wedge V_2 > V_G) \wedge (w_1 \wedge \neg w_2)$
$I_3 = (V_1 > V_G \wedge V_2 \leq V_G) \wedge (\neg w_1 \wedge w_2)$
$I_4 = (V_1 > V_G \wedge V_2 > V_G) \wedge (\neg w_1 \wedge \neg w_2)$
$I_5 = (V_1 + V_2 > V_G \wedge V_1 \leq V_G \wedge V_2 \leq V_G \wedge V_1 \leq V_2) \wedge (\neg w_1 \wedge w_2)$
$I_6 = (V_1 + V_2 > V_G \wedge V_1 \leq V_G \wedge V_2 \leq V_G \wedge V_1 > V_2) \wedge (w_1 \wedge \neg w_2)$

We note the following observation about states in $S$:

**Observation 3.** *For any value of $V_1$, $V_2$ and $V_G$, there exists an assignment to $w_1$ and $w_2$ that changes the state to a state is in $S$.*

The environment can change the values of sensors 1 and 2. In addition, environment can keep the current value of a sensor by self-loop environment transitions. However, environment cannot change the status of switches, change the generated load, or leave the invariant. Thus, the set of environment transitions $\delta_e$ is equal to $\{(s_0, s_1) | \big(w_1(s_0) = w_1(s_1)\big) \wedge \big(w_2(s_0) = w_2(s_1)\big) \wedge \big(V_G(s_0) = V_G(s_1)\big) \wedge \big(\cap_{i=1}^{6} I_i(s_0) \Rightarrow \cap_{i=1}^{6} I_i(s_1)\big)\}$, where $v(s_j)$ shows the value of the variable or predicate $v$ in state $s_j$.

Program cannot change the value of any sensor. Thus, set of program restrictions for this program is $\delta_r = \{(s_0, s_1) |\ V_G(s_0) \neq V_G(s_1) \vee V_1(s_0) \neq V_1(s_1) \vee V_2(s_0) \neq V_2(s_1)\}$.

For the sake of presentation, we also consider the case where that program cannot change the status of more than one switch in one transition. For this case, we add more transitions to the set of program restrictions. We call the set of program restrictions for this case $\delta_{r_2}$ and it is equal to $\{(s_0, s_1) |\ V_G(s_0) \neq V_G(s_1) \vee V_1(s_0) \neq V_1(s_1) \vee V_2(s_0) \neq V_2(s_1) \vee (w_1(s_0) \neq w_1(s_1) \wedge w_2(s_0) \neq w_2(s_1))\}$.

### *4.2. Addition of Stabilization*

Here, we apply Algorithm 1 to add stabilization to program $p$ defined in Section 4.1. We illustrate the result of applying Algorithm 1 for two sets of program restrictions, $\delta_{r_1}$ and $\delta_{r_2}$.

#### 4.2.1. Adding Stabilization for $\delta_{r_1}$

At the beginning of Algorithm 1, $R$ is initialized with $S$. In the first iteration of loop on Lines 4–14, $R_p$ is the set of states outside $S$ that can reach a state in $S$ with only one program transition. A program transition cannot change the value of any sensor. According to Observation 3, from each state in $\neg S$ it is possible to reach a state in $S$ with changing the status of switches. Therefore, the following set of transitions are added to $\delta'_p$ by Line 9:

$$\{(s_0, s_1) |\ V_1(s_0) = V_1(s_1) \wedge V_2(s_0) = V_2(s_1) \wedge V_G(s_0) = V_G(s_1) \wedge s_0 \notin \cup_{i=1}^{6} I_i \wedge s_1 \in \cup_{i=1}^{6} I_i\}$$

Since every state in $\neg S$ ($\neg R$) is in $R_p$, there does not exist any environment transition starting from any state to a state in $\neg(R \cup R_p)$. Therefore, all the states in $\neg R$ are added to $R$ by Line 12. In the second iteration, no more states are added to $R$. Thus, loop on Lines 4–14 terminates. Since there is no state in $\neg R$, the algorithm returns $\delta'_p$.

#### 4.2.2. Adding Stabilization for $\delta_{r_2}$

At the beginning of Algorithm 1, $R$ is initialized with $S$. In the first iteration of loop on Lines 4–14, $R_p$ is the set of states outside $S$ that can reach a state in $S$ with only one program transition. A program transition cannot change the value of any sensor. In addition, according to $\delta_{r_2}$, it cannot change the status of both switches simultaneously. Therefore, state predicate $R_p$ is the union of state predicates $R_{p_1}$ to $R_{p_6}$ as follows ($\oplus$ denotes the xor operation):

$R_{p_1} = (V_1 + V_1 \leq V_G) \wedge (w_1 \oplus w_2)$
$R_{p_2} = (V_1 \leq V_G \wedge V_2 > V_G) \wedge (w_1 \oplus \neg w_2)$
$R_{p_3} = (V_1 > V_G \wedge V_2 \leq V_G) \wedge (\neg w_1 \oplus w_2)$
$R_{p_4} = (V_1 > V_G \wedge V_2 > V_G) \wedge (\neg w_1 \oplus \neg w_2)$
$R_{p_5} = (V_1 + V_2 > V_G \wedge V_1 \leq V_G \wedge V_2 \leq V_G \wedge V_1 \leq V_2) \wedge (\neg w_1 \oplus w_2)$
$R_{p_6} = (V_1 + V_2 > V_G \wedge V_1 \leq V_G \wedge V_2 \leq V_G \wedge V_1 > V_2) \wedge (w_1 \oplus \neg w_2)$

Similarly, $\neg(R \cup R_p)$ includes every state that is outside $S$ and more than one step are needed to reach a state in $S$. Therefore, state predicate $\neg(R \cup R_p)$ is the union of state predicates $R'_{p_1}$ to $R'_{p_6}$ as follows:

$R'_{p_1} = (V_1 + V_1 \leq V_G) \wedge (\neg w_1 \wedge \neg w_2)$
$R'_{p_2} = (V_1 \leq V_G \wedge V_2 > V_G) \wedge (\neg w_1 \wedge w_2)$
$R'_{p_3} = (V_1 > V_G \wedge V_2 \leq V_G) \wedge (w_1 \wedge \neg w_2)$
$R'_{p_4} = (V_1 > V_G \wedge V_2 > V_G) \wedge (w_1 \wedge w_2)$
$R'_{p_5} = (V_1 + V_2 > V_G \wedge V_1 \leq V_G \wedge V_2 \leq V_G \wedge V_1 \leq V_2) \wedge (w_1 \wedge \neg w_2)$
$R'_{p_6} = (V_1 + V_2 > V_G \wedge V_1 \leq V_G \wedge V_2 \leq V_G \wedge V_1 > V_2) \wedge (\neg w_1 \wedge w_2)$

Now, observe that from any state in $R_p$, it is possible to reach a state in $\neg(R \cup R_p)$ by an environment transition. Therefore, no state is added to $R$ in the first iteration and loop on Lines 4–14 terminate in the first iteration. Since all the states outside $S$ remain in $\neg R$, the algorithm declares that there is no solution to the addition problem. Therefore, according to the completeness of Algorithm 1, there does not exist a 2-stabilizing program for the smart grid described in this section when the set of program restriction is $\delta_{r_2}$.

## 5. Addition of Fault-Tolerance

In this section, we present our algorithm for adding failsafe and masking fault-tolerance. In Section 5.1, we identify the problem statement for adding fault-tolerance. In Section 5.2, we present an algorithm for adding failsafe fault-tolerance. The proofs of the soundness and completeness and the complexity results for this algorithm are provided in Appendix A. In Section 5.3, we present an algorithm for adding masking fault-tolerance. The proofs of the soundness and completeness and the complexity results for this algorithm are provided in Appendix B.

### *5.1. Problem Definition*

The problem statement for the addition of fault-tolerance is as follows:

> Given $p$, $\delta_e$, $S$, $spec$, set of program restrictions $\delta_r$, $k > 1$ and $f$ such that $p$ $k$-satisfies $spec$ from $S$ in environment $\delta_e$ and $\delta_p \cap \delta_r = \emptyset$, identify $p'$ and invariant $S'$ such that:
>
> **C1** every computation of $p'[]_k\delta_e$ that starts in a state in $S'$ is a computation of $p[]_k\delta_e$ that starts in $S$ and
> **C2** $p'$ is (failsafe or masking) $k$-$f$-tolerant to $spec$ from $S'$ in environment $\delta_e$ and
> **C3** $\delta'_p \cap \delta_r = \emptyset$

The problem statement requires that the program does not introduce new behaviors in the absence of faults (Constraint **C1**), provides desired fault-tolerance (Constraint **C2**) and does not include any transition in $\delta_r$ (Constraint **C3**).

### 5.2. Algorithm to Add Failsafe Fault-Tolerance

The procedure for adding failsafe fault-tolerance for $k = 2$ is shown in Algorithm 2. In this algorithm, set $ms_1$ is the set of states from which there exists a computation suffix that violates the safety of *spec*. $ms_2$ is the set of states such that, if they are reached by a program or fault transition, or starting from them, there exists a computation suffix that violates safety. Note that $ms_2$ always includes $ms_1$. First, $ms_1$ is initialized to $\{s_0|(s_0, s_1) \in f \cap \delta_b\}$ and $ms_2$ is initialized to $ms_1 \cup \{s_0|\exists s_1 :: (s_0, s_1) \in \delta_e \cap \delta_b\}$ by Lines 1 and 2, respectively. Set $mt$ is the set of transitions that the final program cannot have, as they are in $\delta_b \cup \delta_r$, or reach a state in $ms_2$.

**Algorithm 2** Adding Failsafe Fault-Tolerance

**Input:** $S_p, \delta_p, \delta_e, S, \delta_b, \delta_r$ and $f$
**Output:** $(S', \delta'_p)$ or Not-Possible

1: $ms_1 = \{s_0|\exists s_1 :: (s_0, s_1) \in f \cap \delta_b\}$;
2: $ms_2 = ms_1 \cup \{s_0|\exists s_1 :: (s_0, s_1) \in \delta_e \cap \delta_b\}$;
3: $mt = \{(s_0, s_1)|\ (s_0, s_1) \in (\delta_b \cup \delta_r) \vee s_1 \in ms_2\}$;
4: **repeat**
5: $\quad ms'_1 = ms_1$;
6: $\quad ms'_2 = ms_2$;
7: $\quad ms_1 = ms_1 \cup \{s_0|\ \exists s_1 : s_1 \in ms_2 : (s_0, s_1) \in f\} \cup \{s_0|(\exists s_1 :: (s_1 \in ms_1 \wedge (s_0, s_1) \in \delta_e) \vee (s_0, s_1) \in \delta_e \cap \delta_b)) \wedge (\forall s_2 :: (s_0, s_2) \in mt)\}$;
8: $\quad ms_2 = ms_2 \cup ms_1 \cup \{s_0|\exists s_1 : s_1 \in ms_1 : (s_0, s_1) \in \delta_e)\}$;
9: $\quad mt = \{(s_0, s_1)|\ (s_0, s_1) \in (\delta_b \cup \delta_r) \vee s_1 \in ms_2\}$;
10: **until** $(ms'_1 = ms_1 \wedge ms'_2 = ms_2)$
11: $\delta'_p = \delta_p | S - mt$;
12: $S', \delta'_p = ClosureAndDeadlocks(S - ms_2, \delta'_p, \delta_e)$;
13: **repeat**
14: $\quad$ **if** $S' = \emptyset$ **then**
15: $\quad\quad$ **return** Not-Possible;
16: $\quad$ **end if**
17: $\quad S'' = S'$;
18: $\quad ms_3 = \{s_0|\ (\exists s_1, s_2 :: (s_0, s_1) \in \delta_e\ \wedge (s_0, s_2) \in \delta_p\ ) \wedge (\nexists s_3 :: (s_0, s_3) \in \delta'_p)\}$;
19: $\quad ms_4 = \{s_0|\exists s_1 :: (s_1 \in ms_3 \wedge (s_0, s_1) \in \delta_e)\}$
20: $\quad S', \delta'_p = ClosureAndDeadlocks(S - ms_4, \delta'_p, \delta_e)$;
21: **until** $(S'' = S')$
22: $\delta'_p = \Big(\delta'_p \cup \big((S_p - S') \times S_p\big)\Big) - mt$;
23: **return** $(S', \delta'_p)$;

24: $ClosureAndDeadlocks(S, \delta_p, \delta_e)$
25: $\quad$ **repeat**
26: $\quad\quad S' = S$;
27: $\quad\quad \delta'_p = \delta_p$;
28: $\quad\quad S = S - \{s_0|\ (\forall s_1 : s_1 \in S : (s_0, s_1) \notin \delta_p \cup \delta_e)\}$;
29: $\quad\quad S = S - \{s_0|\ \exists s_1 :: (s_0, s_1) \in \delta_e \wedge s_0 \in S \wedge s_1 \notin S\}$;
30: $\quad\quad \delta_p = \delta_p - \{(s_0, s_1) :: s_0 \in S \wedge s_1 \notin S\}$;
31: $\quad$ **until** $(S' = S) \wedge (\delta'_p = \delta_p)$
32: $\quad$ **return** $S, \delta_p$;

In the loop on Lines 4–10, more states are added to $ms_1$ and $ms_2$. Consequently, we update $mt$. Any state $s_0$ is added to $ms_1$ by Line 7 in two cases: (1) if there exists a fault transition starting from $s_0$ that reaches a state in $ms_2$, or (2) there exists an environment transition $(s_0, s_1)$ such that $(s_0, s_1)$ is a bad transition or $s_1 \in ms_1$ and any other transition starting from $ms_1$ reaches a state in $ms_2$ (i.e., any transition $(s_0, s_2) \in mt$).

A state is added to $ms_2$ by Line 8 if it is added to $ms_1$ or if there exists an environment transition to a state in $ms_1$. We update $mt$ by Line 9 to include transitions to new states added to $ms_2$. The loop on Lines 4–10 terminates if no state is added to $ms_1$ or $ms_2$ in an iteration.

Then, we focus on creating new invariant, $S'$, for the revised program. $S'$ cannot include any transition in $ms_2$, as starting from any state in $ms_2$, there is a computation which violates the safety. In addition, the set of program transitions of the revised program, $\delta'_p$, cannot include any transition in $mt$, as by any transition in $mt$ a state in $ms_2$ is reached. Thus, we initialized $\delta'_p$ with $\delta_p|S - mt$. Note that $S'$ must be closed in $p' \cup \delta_e$ and it cannot include any deadlock state. (Note, as we said in Section 2, for original deadlock states we have added self-loops. Thus, any deadlock state at this point is created because of removing transitions. Having these deadlock state in the invariant create new behavior inside invariant which is not acceptable.)

Thus, whenever we remove a state from $S'$ we ensure that the $S'$ is closed in $p' \cup \delta_e$ and does not include any deadlock state by Line 12.

To satisfy condition **C1**, in the loop on Lines 13–21, we remove certain states that cause new computations for $p'$ that are not computations of $p$. Suppose starting from $s_0$ there exists environment transition $(s_0, s_1)$. In addition, there exists a program transition $(s_0, s_2)$ in the set of program transitions of the original program, $\delta_p$. Set $ms_3$ includes any state like $s_0$. If $s_0$ is reached by environment transition $(s_3, s_0)$, in the original program, according to the fairness assumption, $(s_0, s_1)$ cannot occur. Thus, sequence $\langle s_3, s_0, s_1 \rangle$ cannot be in any $p[]_2\delta_e$ computation. However, if we remove program transition $(s_0, s_2)$ in the revised program, $\langle s_3, s_0, s_1 \rangle$ can be in a $p'[]_2\delta_e$ computation. Therefore, we have to remove any state like $s_3$ (i.e., all states in $ms_4$ from the invariant (Line 20)).

After creating invariant $S'$, we add program transitions outside it to $\delta'_p$. Note that outside $S'$, any program transition which is not in $mt$ is allowed to be in the final program. In Line 15, the algorithm declares that no solution to the addition problem exists, if $S'$ is empty. Otherwise, at the end of the algorithm, it returns $(\delta'_p, S')$ as the solution to the addition problem.

### *5.3. Algorithm to Add Masking Fault-Tolerance*

In this section, we present an algorithm for adding masking fault-tolerance in the presence of unchangeable environment actions. The intuition behind this algorithm is as follows: first, we utilize the ideas from adding stabilization. Intuitively, in Algorithm 1, we constructed the set $R$ from where recovery to the invariant ($S$) was possible. In the case of stabilization, we wanted to ensure that $R$ includes all states. However, for masking fault-tolerance recovery from all states is not necessary. We also need to ensure that recovery from $R$ is not prevented by faults. This may require us to prevent the program from reaching some states in $R$. Hence, this process needs to be repeated to identify a set $R$ such that recovery to $S$ is provided and faults do not cause the program to reach a state outside $R$. In addition, in masking fault-tolerance, like failsafe fault-tolerance, the program must satisfy safety of specification even in presence of faults. Thus, the details of Algorithm 3 are as follows.

We start with an invariant equal to the original invariant (i.e., $S' = S$). Like Algorithm 1, we set $\delta'_p$ to $\delta_p|S$ which is the set of all program transitions inside the invariant. Like in the case of Algorithm 2, set $ms_1$ includes all states such that no matter how they are reached, there is a computation that is not desirable. In the case of failsafe fault-tolerance, such computation violates safety. On the other hand, here in case of masking fault-tolerant, such computation either violates safety or never reaches the invariant. $ms_2$ is the set of states such that, if they are reached by a program or fault transition, or starting from them, there exists a computation suffix which either violates safety or never reaches the invariant. Same as Algorithm 2, set $mt$ always includes all transitions to states in $ms_2$. We initialize sets $ms_1$, $ms_2$ and $mt$ as done in Algorithm 2 and expand them in the loop on Lines 6–43. In this loop, we use Algorithm 1 and Algorithm 2 with some modification.

**Algorithm 3** Adding Masking Fault-Tolerance

**Input:** $S_p, \delta_p, \delta_e, S, \delta_b, \delta_r$, and $f$
**Output:** $(S', \delta'_p)$ or Not-Possible

1: $S' = S$;
2: $\delta'_p = (\delta_p | S)$;
3: $ms_1 = \{s_0 | (s_0, s_1) \in f \cap \delta_b\}$;
4: $ms_2 = ms_1 \cup \{s_0 | \exists s_1 : (s_0, s_1) \in \delta_e \cap \delta_b\}$;
5: $mt = \{(s_0, s_1) |\ (s_0, s_1) \in (\delta_b \cup \delta_r) \vee s_1 \in ms_2\}$;
6: **repeat**
7: $\quad ms'_1 = ms_1$;
8: $\quad ms'_2 = ms_2$;
9: $\quad R = S'$;
10: $\quad S'' = S'$;
11: $\quad R_p = \{\}$;
12: $\quad$ **repeat**
13: $\quad\quad R' = R$;
14: $\quad\quad R'_p = \{s_0 | s_0 \notin R \wedge \exists s_1 : s_1 \in R : (s_0, s_1) \notin mt\}$;
15: $\quad\quad R_p = R'_p \cup Rp$;
16: $\quad\quad$ **for** *each* $s_0 \in R'_p$ **do**
17: $\quad\quad\quad \delta'_p = \delta'_p \cup \{(s_0, s_1) | (s_0, s_1) \notin mt \wedge s_1 \in R\}$;
18: $\quad\quad$ **end for**
19: $\quad\quad$ **for** *each* $s_0 \notin R : \Big(\nexists s_2 : s_2 \in (\neg(R \cup R_p) \cup ms_1) : (s_0, s_2) \in \delta_e\Big) \wedge \Big(\big(\exists s_1 : s_1 \in (R \cup R_p) - ms_1 : (s_0, s_1) \in \delta_e\big) \wedge (\nexists s_2 :: (s_0, s_2) \in (\delta_e \cap \delta_b))\big) \vee s_0 \in R_p\Big)$ **do**
20: $\quad\quad\quad R = R \cup s_0$;
21: $\quad\quad$ **end for**
22: $\quad$ **until** $(R' = R)$;
23: $\quad ms_1 = ms_1 \cup \neg(R \cup R_p)$;
24: $\quad ms_2 = ms_2 \cup \neg R$;
25: $\quad$ **repeat**
26: $\quad\quad ms''_1 = ms_1$;
27: $\quad\quad ms''_2 = ms_2$;
28: $\quad\quad ms_1 = ms_1 \cup \{s_0 |\ \exists s_1 : s_1 \in ms_2 : (s_0, s_1) \in f\} \cup \{s_0 | \big(\exists s_1 : s_1 \in ms_1 : (s_0, s_1) \in \delta_e\big) \vee (s_0, s_1) \in (\delta_e \cap \delta_b)) \wedge \big(\nexists s_2 :: (s_0, s_2) \in \delta'_p\big)\}$;
29: $\quad\quad ms_2 = ms_2 \cup ms_1 \cup \{s_0 | \exists s_1 : s_1 \in ms_1 : (s_0, s_1) \in \delta_e)\}$;
30: $\quad\quad mt = \{(s_0, s_1) |\ (s_0, s_1) \in (\delta_b \cup \delta_r) \vee s_1 \in ms_2\}$;
31: $\quad$ **until** $ms''_1 = ms_1 \wedge ms''_2 = ms_2$
32: $\quad \delta'_p = \delta'_p - mt$;
33: $\quad S', \delta'_p = ClosureAndDeadlocks(S' - ms_2, \delta'_p, \delta_e)$;
34: $\quad$ **repeat**
35: $\quad\quad$ **if** $S' = \phi$ **then**
36: $\quad\quad\quad$ **return** Not-Possible;
37: $\quad\quad$ **end if**
38: $\quad\quad S''' = S'$;
39: $\quad\quad ms_3 = \{s_0 |\ \big(\exists s_1, s_2 :: (s_0, s_1) \in \delta_e \wedge (s_0, s_2) \in \delta_p\ \big) \wedge \big(\nexists s_3 : (s_0, s_3) \in \delta'_p\big)\}$;
40: $\quad\quad ms_4 = \{s_0 | \exists s_1 :: (s_1 \in ms_3 \wedge (s_0, s_1) \in \delta_e)\}$
41: $\quad\quad S', \delta'_p = ClosureAndDeadlocks(S - ms_4, \delta'_p, \delta_e)$;
42: $\quad$ **until** $(S''' = S')$
43: **until** $(S'' = S' \wedge ms'_1 = ms_1 \wedge ms'_2 = ms_2)$
44: **return** $(S', \delta'_p)$ ;

In the loop on Lines 12–22 we build set $R$ which includes all states from which all computations reach a state in $S'$. In addition, all required program transitions are added to $\delta'_p$ by Line 17. When loop on Lines 12–22 terminates, we add all states in $\neg(R \cup R_p)$ to $ms_1$. We also set $ms_2$ to $\neg R$. Then, in the

loop on Lines 25–31 we expand $ms_1$ and $ms_2$ with the same procedure used in Algorithm 2. In Line 32, we remove any transition in $mt$ from $\delta'_p$. In Line 33, we remove all states in $ms_2$ from $S'$ and ensure closure and deadlock freedom.

In the loop on Lines 34–42, we remove some states from $S'$ to avoid new behavior inside the invariant just like what we did in Algorithm 2. If any state $s_0$ is removed from $S'$ in Lines 33 or 41, we need to repeat the loop on Lines 6–43, because it is possible that a state in $R$ was dependent on $s_0$ to reach $S'$ but $s_0$ is not in $S'$ anymore. We also need to repeat this loop if the set of $ms_1$ or $m_2$ has changed. In Line 36, the algorithm declares that there does not exist a solution if $S'$ is empty. Otherwise, when loop on Lines 6–43 terminates, the algorithm returns $(\delta'_p, S')$ as the solution to the addition problem.

We note that the addition of nonmasking fault-tolerance considered [6] is also possible with Algorithm 3. In particular, in this case, we need to set $\delta_b$ to be the empty set. In principle, Algorithm 3 could also be used to add stabilization. However, we presented Algorithm 1 separately since it is a much simpler algorithm and forms the basis of Algorithm 3.

## 6. Experimental Results

In this section, we provide some of our experimental results of applying algorithms provided in previous sections. We have implemented the algorithms proposed in this paper in Java in a package called JavaRepair. JavaRepair and the code for examples of this section can be downloaded from [4]. JavaRepair uses JavaBDD [4] for symbolic repair using Binary Decision Diagrams (BDDs). All experiments are done on a 64-bit Windows 8.1 machine with AMD A8-6410 APU 2.00 GHz CPU and 8 GB RAM. We first provide our results for the addition of stabilization in Section 6.1, next we focus on the addition of fault-tolerance in Section 6.2.

### *6.1. Addition of Stabilization*

In this section we use the JavaRepair package to repair/synthesis stabilizing programs for two variations of the shepherding problem [10].

#### 6.1.1. One-Dimensional Stabilizing Shepherding Problem

In this section, we focus on the shepherding problem introduced in Section 1. We first consider the one-dimensional version of the problem where both farmer and the sheep move in a line. Thus, we have a row of $n$ cells that the sheep/farmer can be in. Thus, we have two variables $c_f$ and $c_s$ which represent the location of the farmer and the sheep, respectively (see Figure 5). The farmer and the sheep can change their location at most by one cell at a time.

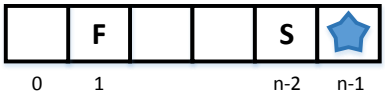


**Figure 5.** An example state of an 1D shepherding problem with $n = 6$. In this example, $c_f = 1$ and $c_s = n - 2$.

The rightmost cell is marked as the desired location. The farmer wants to steer the sheep to this cell. Thus, the invariant for the problem is $\{s | c_s(s) = n - 1\}$, where $n$ is the number of cells in the row. The sheep always tries to increase its distance with the farmer. When the farmer and the sheep are in the same location, the sheep non-deterministically decides to go to the right or left, if it has space in

its both left and right. When the sheep is in the desired location, it stops moving further. With this explanation, the set of environment transitions is

$$\begin{aligned}\delta_e = \Big\{&(s_0, s_1) | c_s(s_0) \neq n-1 \wedge c_f(s_0) = c_f(s_1) \wedge \\ &\Big( \big(c_f(s_0) < c_s(s_0) \wedge c_s(s_0) < n-1 \wedge c_s(s_1) = c_s(s_0)+1\big) \vee \\ &\big(c_f(s_0) > c_s(s_0) \wedge c_s(s_0) > 0 \wedge c_s(s_1) = c_s(s_0)-1\big) \vee \\ &\big(c_f(s_0) = c_s(s_0) \wedge c_s(s_0) < n-1 \wedge c_s(s_1) = c_s(s_0)+1\big) \vee \\ &\big(c_f(s_0) = c_s(s_0) \wedge c_s(s_0) > 0 \wedge c_s(s_1) = c_s(s_0)-1\big)\Big)\Big\}\end{aligned}$$

Also, since the farmer cannot move more than one cell in one step, the set of program restrictions is

$$\delta_r = \{(s_0, s_1) |\ |c_f(s_0) - c_f(s_1)| > 1\}.$$

Because of the non-determinism in the movement of the sheep, a program for the farmer can go into an infinite loop such that it never reaches the goal state. To understand how that can happen, consider this scenario: suppose both farmer and sheep are in the cell 0. Since the left side of the sheep is closed, it can only move to the right. Thus, it moves to the second cell. Now, suppose the farmer also decides to go to the second cell. Thus, again both of them are in the same cell. However, this time, since the left side of the sheep is open, it may decide to go there. Suppose sheep decides to go the left cell. Then, the farmer also goes back to the first cell. Once again, they are both in the first cell and the same scenario can repeat forever. A correct program for the farmer utilizes the behavior of the sheep, while avoiding going into an infinite loop and steers the sheep to the desired location.

We modeled this problem using JavaRepair for different sizes. We used a program with an empty set of transitions as the input program. Algorithm 1 successfully adds stabilization to the given program. Table 1 shows the repair time for different sizes.

#### 6.1.2. Two-Dimensional Stabilizing Shepherding Problem

In this section, we consider the shepherding problem described in Section 6.1.1 in a two-dimensional space. Thus, instead of a single row of cells, we have a grid of cells where the sheep and the farmer can move. Thus, we have four variables $r_f$, $c_f$, $r_s$ and $c_s$ that represent the row and column of the farmer and the sheep, respectively (see Figure 6).

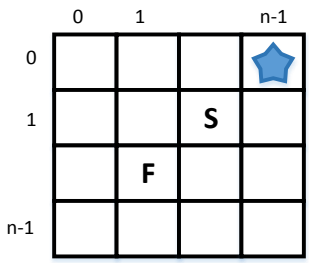


**Figure 6.** An example state of a 2D shepherding problem with $n = 4$. In this example, $r_f = 2$, $c_f = 1$, $r_s = 1$ and $c_s = 2$.

The upper right corner of the grid is the desired location. Thus, the invariant for this problem is $\{s | r_s(s) = 0 \wedge c_s(s) = n-1\}$, where again $n$ is the number of cells in a row. The sheep again tries to increases its distance with the farmer. The sheep and the farmer can only change either their row or

column by at most one at each step. As in Section 6.1.1, sheep stops moving once it is in the desired location. With this explanation, the set of environment transitions is

$$\begin{aligned}\delta_e = \Big\{ & (s_0, s_1) | \neg (r_s(s_0) = 0 \wedge c_s(s_0) = n-1) \wedge \\ & (r_f(s_0) = r_f(s_1) \wedge c_f(s_0) = c_f(s_1)) \wedge \\ & (c_s(s_0) = c_s(s_1) \vee r_s(s_0) = r_s(s_1)) \wedge \\ & \Big( (c_f(s_0) < c_s(s_0) \wedge c_s(s_0) < n-1 \wedge c_s(s_1) = c_s(s_0)+1) \vee \\ & (c_f(s_0) > c_s(s_0) \wedge c_s(s_0) > 0 \wedge c_s(s_1) = c_s(s_0)-1) \vee \\ & (c_f(s_0) = c_s(s_0) \wedge c_s(s_0) < n-1 \wedge c_s(s_1) = c_s(s_0)+1) \vee \\ & (c_f(s_0) = c_s(s_0) \wedge c_s(s_0) > 0 \wedge c_s(s_1) = c_s(s_0)-1) \vee \\ & (r_f(s_0) < r_s(s_0) \wedge r_s(s_0) < n-1 \wedge r_s(s_1) = r_s(s_0)+1) \vee \\ & (r_f(s_0) > r_s(s_0) \wedge r_s(s_0) > 0 \wedge r_s(s_1) = r_s(s_0)-1) \vee \\ & (r_f(s_0) = r_s(s_0) \wedge r_s(s_0) < n-1 \wedge r_s(s_1) = r_s(s_0)+1) \vee \\ & (r_f(s_0) = r_s(s_0) \wedge r_s(s_0) > 0 \wedge r_s(s_1) = r_s(s_0)-1) \Big) \Big\}\end{aligned}$$

Also, the set of program restrictions is

$$\delta_r = \{(s_0, s_1) |\ |c_f(s_0) - c_f(s_1)| + |r_f(s_0) - r_f(s_1)| > 1\}.$$

Like one-dimensional case discussed in Section 6.1.1, with a poorly designed farmer, here is also possible to chase the sheep forever and never reach the desired state. Table 1 shows the repair time using JavaRepair for this problem for different sizes.

**Table 1.** Average Time of Addition of Stabilization for Shepherding Problem.

| Size | Addition Time (s) | |
|---|---|---|
| | 1D Shepherding | 2D Shepherding |
| 4 | 0.008 | 0.025 |
| 5 | 0.023 | 0.119 |
| 6 | 0.007 | 0.231 |
| 7 | 0.007 | 0.164 |
| 8 | 0.007 | 0.115 |
| 9 | 0.014 | 0.656 |
| 10 | 0.038 | 1.141 |
| 11 | 0.013 | 1.310 |
| 12 | 0.011 | 0.802 |
| 13 | 0.010 | 1.952 |
| 14 | 0.012 | 2.776 |
| 15 | 0.012 | 2.514 |
| 16 | 0.010 | 0.680 |
| 17 | 0.024 | 7.636 |
| 18 | 0.029 | 16.826 |
| 19 | 0.026 | 16.361 |
| 20 | 0.020 | 9.655 |
| 30 | 0.128 | 42.137 |
| 40 | 0.119 | 46.789 |

### *6.2. Addition of Fault-Tolerance*

In this section, we use the JavaRepair package to repair/synthesis fault-tolerant programs for two other variations of the shepherding problem. We first create a failsafe fault-tolerant program in Section 6.2.1. Next, we create a masking fault-tolerant program in Section 6.2.2.

#### 6.2.1. Failsafe Fault-Tolerant Shepherding Problem

For this variation, we change the two-dimensional version described in Section 6.1.2 as follows: there is no requirement of steering the sheep to the desired location. Instead, we require that the farmer must be always close to the sheep. Specifically, we require that the farmer and sheep can be far apart at most by one row or one cell. We model this requirement as a safety property with the set of bad transitions

$$\begin{aligned}&\delta_b = \Big\{(s_0, s_1)| \\ &\neg\Big(\big(r_s(s_1) = r_f(s_1) \wedge c_s(s_1) = c_f(s_1)\big)\vee \\ &\big(|r_s(s_1) - r_f(s_1)| = 1 \wedge c_s(s_1) = c_f(s_1)\big)\vee \\ &\big(r_s(s_1) = r_f(s_1) \wedge |c_s(s_1) - c_f(s_1)| = 1\big)\Big)\Big\}.\end{aligned}$$

The invariant, the set of environment transitions $\delta_e$ and the set of program restrictions $\delta_r$ are the same as those for two-dimensional case explained in Section 6.1.2. For addition of fault-tolerance we have a set of faults as the input. We consider two sets of faults and try to repair the program with each of them. The first set captures faults that can change the state of the program arbitrary, that is, for any state the program may change to any other state. We denote this set by $f_{arbitrary}$. The second set of fault transitions that we consider captures faults that cause the system transition to a state where both farmer and sheep co-locate in one of the cells except the desired cell. We model this transition as a set of fault transitions

$$\begin{aligned}&f_{same-location} = \big\{(s_0, s_1)| \\ &r_s(s_0) = 0 \wedge c_s(s_0) = n-1 \wedge \\ &r_s(s_1) = r_f(s_1) \wedge c_s(s_1) = c_f(s_1) \wedge \\ &\neg(r_s(s_1) = 0 \wedge c_s(s_1) = n-1)\big\}\end{aligned}$$

We modeled this problem using JavaRepair for different sizes. The Algorithm 2 does not find any solution for the case of arbitrary faults, $f_{arbitrary}$. From the completeness of Algorithm 2, we know that there is no solution for this case. On the other hand, the algorithm finds a solution for $f_{same-location}$. Table 2 shows the repair time for different sizes.

#### 6.2.2. Masking Fault-Tolerant Shepherding Problem

In this section, we add the requirement of steering the sheep to the desired location to the failsafe version explained in Section 6.2.1. Note that this requirement is weaker than the one specified in the stabilization version explained in Section 6.1.2 where we required that the farmer must steer the sheep to the desired location starting from any *arbitrary state*. On the other hand, in this section, we require that the farmer should steer the sheep to the goal state once the program starts in states reached by the fault transitions (i.e., fault-span). The invariant, the set of environment transitions and the set of program restrictions are the same as those defined in Section 6.2.1. Like Section 6.2.1, we consider two sets of fault transitions $f_{arbitrary}$ and $f_{same-location}$. Like failsafe case, there is no solution for arbitrary faults but for same-location faults, Algorithm 3 finds a solution. Table 2 shows the addition time for this problem.

**Table 2.** Average Time of Addition of Fault-tolerance for Shepherding Problem.

| Size | Addition Time (s) | | | |
|---|---|---|---|---|
| | Failsafe | | Masking | |
| | Arbitrary Fault | Same-Location Fault | Arbitrary Fault | Same-Location Fault |
| 4 | 0.006 | 0.009 | 0.009 | 0.057 |
| 5 | 0.019 | 0.027 | 0.043 | 0.176 |
| 6 | 0.006 | 0.018 | 0.023 | 0.129 |
| 7 | 0.007 | 0.017 | 0.016 | 0.197 |
| 8 | 0.004 | 0.012 | 0.013 | 0.181 |
| 9 | 0.019 | 0.027 | 0.025 | 0.340 |
| 10 | 0.014 | 0.029 | 0.020 | 0.424 |
| 11 | 0.007 | 0.024 | 0.018 | 0.547 |
| 12 | 0.004 | 0.033 | 0.015 | 0.493 |
| 13 | 0.005 | 0.035 | 0.016 | 0.698 |
| 14 | 0.005 | 0.021 | 0.018 | 0.807 |
| 15 | 0.006 | 0.019 | 0.025 | 0.928 |
| 16 | 0.004 | 0.015 | 0.014 | 0.735 |
| 17 | 0.032 | 0.081 | 0.074 | 1.142 |
| 18 | 0.017 | 0.281 | 0.116 | 1.273 |
| 19 | 0.019 | 0.284 | 0.036 | 1.416 |
| 20 | 0.014 | 0.049 | 0.099 | 1.245 |
| 30 | 0.886 | 1.137 | 0.973 | 4.995 |
| 40 | 1.204 | 1.287 | 1.246 | 5.232 |

## 7. Discussion and Extensions of Algorithms

In this section, we consider problems related to those addressed in Sections 3 and 5. Our first variation focuses on Definition 6. In this definition, we assumed that the environment is *fair*. Specifically, at least $k-1$ actions execute between any two environment actions. We consider variations where (1) this property is satisfied eventually. In other words, for some initial computation, environment actions may prevent the program from executing. However, eventually, fairness is provided to program actions and (2) program actions are given even with reduced fairness. Specifically, we consider the case where several environment actions can execute in a row but program actions execute infinitely often.

Our second variation is related to the invariant of the revised program, $S'$ and the invariant of the original program, $S$. In the case of adding stabilization, we considered $S' = S$ whereas, in the case of adding fault-tolerance, we considered $S' \subseteq S$.

**Changes for adding stabilization and fault-tolerance with an eventually fair environment.** No changes are required to Algorithm 1 even if the environment is eventually fair. This is due to the fact that this algorithm constructs programs that provide recovery from *any* state, that is, it will provide recovery from the state reached after the point when fairness is restored. For Algorithm 2 and Algorithm 3, we should change the input $f$ to include $\delta_e \cup \delta_f$. The resulting algorithm will ensure that the generated program will allow unfair execution of the program in initial states. However, fault-tolerance will be provided when the fairness is restored.

**Changes for adding stabilization and fault-tolerance with multiple consecutive environment actions.** If environment actions can execute consecutively, we can change input $\delta_e$ to be its transitive closure. In other words, if $(s_0, s_1)$ and $(s_1, s_2)$ are transitions in $\delta_e$, we add $(s_0, s_2)$ to $\delta_e$. With this change, the constructed program will provide stabilization or fault-tolerance even if environment transitions can execute consecutively.

**Changes for adding stabilization and fault-tolerance based on relation between $S'$ (invariant of the fault-tolerant program) and $S$ (invariant of the fault-intolerant program).** No changes are required to Algorithm 1 even if we change the problem statement to allow $S' \subseteq S$ without affecting soundness or completeness. Regarding soundness, observe that the program generated by this algorithm ensures $S' = S$. Hence, it trivially satisfies $S' \subseteq S$. Regarding completeness, the intuition

is that if it was impossible to recover to states in $S$ then it is impossible to recover to states that are a subset of $S$. Regarding Algorithm 2 and Algorithm 3, if $S'$ is required to be equal to $S$ then they need to be modified as follows: in these algorithms if any state $S$ is removed (due to it being in $ms_2$, deadlocks, etc.) then they should declare failure.

**Other applications of our algorithms.** In this paper, we considered the shepherding application as a way to illustrate the role of environment actions. The shepherding program is an instance of several other programs. For example, consider a smart cruise controller. In this example, we have two cars, $A$ and $B$ where car $A$ is the car in front and car $B$ is trying to match the speed of car $A$ and maintain a safe distance. Now, considering from the perspective of car $B$, actions of car $A$ are environment actions that are unchangeable. These actions are essential (without these actions, car $B$ will not be able to satisfy its objectives) and disruptive. In other words, car $A$ plays the same role as the sheep and car $B$ plays the same role as the farmer.

It is also applicable in an example such as merging on a highway. In this example (when viewed from the perspective of the car that is merging), actions of other cars are the same as the role played by sheep in the shepherding problem. More generally, if we have a system with multiple components/processes where only some components are changeable, then these components would be modeled as farmer actions whereas unchangeable actions play the role of the sheep.

## 8. Related Work

This paper focuses on the addition of fault-tolerance properties in the presence of unchangeable environment actions. This problem is an instance of model repair where some existing model/program is repaired to add new properties such as safety, liveness, fault-tolerance and so forth. Model repair with respect to CTL properties was first considered in [11] and abstraction techniques for the same are presented in [12]. Previously, Bonakdarpour et al. [13] has considered the problem of model repair for UNITY specifications [14]. These results identify complexity results for adding properties such as invariant properties, leads-to properties and so forth. Repair of probabilistic algorithms has also been considered in the literature [15]. Roohitavaf and Kulkarni [16] provide repair algorithms for cases where the new desired properties are in conflict with existing properties.

The problem of adding fault-tolerance to an existing program has been discussed in the absence of environment actions. This work includes work on controller synthesis [17–20]. A tool for automated addition of fault-tolerance to distributed programs is presented in [1]. This work utilizes BDD based techniques to enable the synthesis of programs with state space exceeding $10^{100}$. However, this work does not include the notion of environment actions that cannot be removed. Hence, applying it in contexts where some processes/components cannot be changed will result in unacceptable solutions. Model repair for distributed programs using a two-phase lazy approach is proposed in [21].

The work on game theory [22–24] has focused on the problem of repair with the 2-player game where the actions of the second player are not changed. However, this work does not address the issue of fault-tolerance. Also, the role of the environment in our work is more general than that in [22–24]. Specifically, in the work on game theory, it is assumed that the players play in an alternating manner. By contrast, we consider more general interaction with the environment. In [25], authors have presented an algorithm for adding recovery to component-based models. They consider the problem where we cannot add to the interface of a physical component. However, it does not consider the issue of unchangeable actions considered in our work.

Hajisheykhi et al. [26,27] focus on the notion of auditable events which captures special events for which the program must first transition to a special state called auditable state where every process is aware of the auditable events, before returning to the normal states. Thus, compared with fault-span considered in this paper (cf. Definition 14), the set of states that a program may reach other than its states includes states that can be reached by fault transitions and those that can be reached by auditable transitions. Considering the problem of auditable restoration in presence of environment actions is an interesting future work.

## 9. Conclusions

In this paper, we focused on the problem of adding fault-tolerance to an existing program which consists of some actions that are unchangeable. These unchangeable actions arise due to interaction with the environment, inability to change parts of the existing program, constraints on physical components in a cyber-physical system and so on. We presented algorithms for adding stabilization and fault-tolerance. These algorithms are sound and complete and run in polynomial time (in the state space).

We considered the cases where (1) all fault-free behaviors are preserved in the fault-tolerant program, or (2) only a nonempty subset of fault-free behaviors are preserved in the fault-tolerant program. We also considered the cases where (1) environment actions can execute with any frequency for an initial duration and (2) environment actions can execute more frequently than programs. In all these cases, we demonstrated that our algorithm can be extended while preserving soundness and completeness. We provided a Java package including the implementation of our algorithm using BDDs. We presented some of our experimental results of adding stabilization and fault-tolerance using JavaRepair.

There are several future extensions to this work. One of the extensions has to deal with partial observability in distributed or embedded systems. Algorithms in this paper need to be revised by taking into account scenarios where, due to partial observability, one transition added or removed corresponds to a group of transitions [28]. Another extension is to extend the JavaRepair package so that it can be used efficiently and with a reduced learning curve. Yet another extension is to deal with the issue of time in embedded and cyber-physical systems. Specifically, in a CPS, we expect physical components to execute at a certain speed. In other words, an action of the physical component can be thought of to take some time between $\epsilon_1$ and $\epsilon_2$. In turn, this affects the number of steps a computational component can execute in the middle. Note that this maps to the value of $k$ used in our algorithms. Also, our definition of computation requires that the environment cannot execute too frequently. We also intend to extend to cases where a similar restriction also applies to computational components.

**Author Contributions:** Supervision, S.K.; Writing—original draft, M.R.

**Funding:** This work is supported in part by NSF CPS 1329807, NSF SaTC 1318678, and NSF XPS 1533802.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| MDPI | Multidisciplinary Digital Publishing Institute |
| DOAJ | Directory of open access journals |
| TLA | Three letter acronym |
| LD | linear dichroism |

## Appendix A. Soundness, Completeness and Complexity of Algorithm 2

### *Appendix A.1. Soundness*

The following lemma splits condition **C1** into easily verifiable conditions that assist in proving the soundness of Algorithm 2.

**Lemma A1.** *The condition **C1** in the problem definition of addition of fault-tolerance is satisfied for $k = 2$ if conditions below are satisfied:*

1. $S' \subseteq S$
2. $\delta'_p|S' \subseteq \delta_p|S$

3. $\forall s_1 : \left(\exists s_0, s_2, s_3 : s_0 \in S' \wedge (s_0, s_1), (s_1, s_2) \in \delta_e \wedge (s_1, s_3) \in \delta_p\right) : \left(\exists s_4 :: (s_1, s_4) \in \delta'_p\right)$
4. $\forall s_0 :: (\exists s_1 :: (s_0, s_1) \in \delta_p \cup \delta_e)$

**Proof.** We show by induction that if all conditions of the lemma are satisfied, then every prefix of any $p'[]_2\delta_e$ computation that starts from a state in $S'$ is a prefix of a $p[]_2\delta_e$ computation which starts in $S$.
**Base case:** Let $\langle s_0 \rangle$ be the prefix of a $p'[]_2\delta_e$ computation that starts from $S'$. Since $S' \subseteq S$, $s_0 \in S$. Thus, $\langle s_0 \rangle$ is also a prefix of a $p[]_2\delta_e$ computation that starts from $S$.
**Induction hypothesis:** The theorem holds for $\langle s_0, s_1, \ldots, s_i \rangle$.
**Induction step:** We show theorem holds for $\langle s_0, s_1, \ldots, s_{i+1} \rangle$. Since $p' \cup \delta_e$ is closed in $S'$, we know that $s_{i+1} \in S'$. We have two cases for $(s_i, s_{i+1})$:

**Case 1** $(s_i, s_{i+1}) \in \delta'_p$
From $\delta'_p|S' \subseteq \delta_p|S'$, we have $(s_i, s_{i+1}) \in \delta_p$. Therefore, if $\langle s_0, s_1, \ldots, s_i \rangle$ is a prefix of $p[]_2\delta_e$, then $\langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle$ is a prefix of $p[]_2\delta_e$.

**Case 2** $(s_i, s_{i+1}) \in \delta_e$ :
Two sub-cases are possible below this case:

**Case 2.1** $(s_{i-1}, s_i) \in \delta'_p$
In this case, as we have reached $s_i$ by a program transition, even with the fairness assumption, $(s_i, s_{i+1})$ can occur in $p[]_2\delta_e$, Therefore, if $\langle s_0, s_1, \ldots, s_i \rangle$ is a prefix of $p[]_2\delta_e$, then $\langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle$ is a prefix of $p[]_2\delta_e$.

**Case 2.2** $(s_{i-1}, s_i) \in \delta_e$
In this case, there does not exist $s'$ such that $(s_i, s') \in \delta'_p$ (otherwise, because of fairness $(s_i, s_{i+1})$ cannot be in any prefix of $p'[]_2\delta_e$). Now, according to condition 3, we know there does not exist state $s''$ such that $(s_i, s'') \in \delta_p$. Thus, even with the fairness assumption $(s_i, s_{i+1})$ can occur in $p[]_2\delta_e$. Therefore, if $\langle s_0, s_1, \ldots, s_i \rangle$ is a prefix of $p[]_2\delta_e$, then $\langle s_0, s_1, \ldots, s_i, s_{i+1} \rangle$ is a prefix of $p[]_2\delta_e$.

Since every prefix of $p'[]_2\delta_e$ that starts from a state in $S'$ is a prefix of $p[]_2\delta_e$ which starts in $S$, **C1** is satisfied. □

**Theorem A1.** *Algorithm 2 is sound.*

**Proof.** To show the soundness of our algorithm, we need to show that the three conditions of the addition problem are satisfied.

**C1**: Consider a computation $c$ of $p'[]_2\delta_e$ that starts from a state in $S'$. By construction, $c$ starts from a state in $S$ and $\delta'_p|S'$ is a subset of $\delta_p|S'$. Therefore, the first two requirements of Lemma A1 are satisfied. Regarding the third requirement of Lemma A1, suppose that there exists $s_1$ in $S'$ such that $\left(\exists s_0, s_2, s_3 : s_0 \in S' \wedge (s_0, s_1), (s_1, s_2) \in \delta_e \wedge (s_1, s_3) \in \delta_p\right)$ but $\nexists s_4 :: (s_1, s_4) \in \delta'_p$. From $\exists s_2, s_3 :: (s_1, s_2) \in \delta_e \wedge (s_1, s_3) \in \delta_p$ and $\nexists s_4 :: (s_1, s_4) \in \delta'_p$ we can conclude that $s_1$ is in $ms_3$. Then, from $(s_0, s_1) \in \delta_e$ , we know that $s_0$ is in $ms_4$, which is contradiction as $s_0$ is in $S$. Finally, the fourth condition is satisfied by the fact that $S'$ does not include any deadlock state. Since all conditions of Lemma A1 are satisfied, condition *C1* holds.

**C2**: We need to show that $p'$ is a failsafe fault-tolerant revision for $p$. Thus, we need to show constraints of Definition 15 are satisfied.

From *C1*, $S' \subseteq S$, the assumption that $p[]_2\delta_e$ satisfies $spec$ from $S$, $S' \neq \emptyset$ and $S'$ is closed in $p' \cup \delta_e$, all constrains of Definition 10 are satisfied. Thus, $p'[]_2\delta_e$ 2-satisfies $spec$ from $S'$.

Let $spec$ be $\langle Sf, Lv \rangle$. Consider prefix $c$ of $p'[]_2\delta_e[]f$ such that $c$ starts from a state in $S'$. If $c$ does not satisfy $Sf$ then there exists a prefix of $c$, say $\langle s_0, s_1, \ldots, s_n \rangle$, such that it has a transition in $\delta_b$. W.l.o.g., let $\langle s_0, s_1, \cdots, s_n \rangle$ be the smallest such prefix. It follows that $(s_{n-1}, s_n) \in \delta_b$, hence, $(s_{n-1}, s_n) \in mt$. By

construction, $p'$ does not contain any transition in $mt$. Thus, $(s_{n-1}, s_n)$ is a transition of $f$ or $\delta_e$. If it is in $f$ then $s_{n-1} \in ms_1$ (i.e., $s_{n-1} \in ms_2$). If it is in $\delta_e$ then $s_{n-1} \in ms_2$. Therefore, in both cases, $s_{n-1} \in ms_2$ and $(s_{n-2}, s_{n-1}) \in mt$. Again, by construction, we know that $\delta'_p$ does not contain any transition in $mt$, so $(s_{n-2}, s_{n-1})$ is either in $f$ or $\delta_e$. If it is in $f$ then $s_{n-2} \in ms_1$ (i.e., $s_{n-2} \in ms_2$). If it is in $\delta_e$ two cases are possible:

(1) $(s_{n-1}, s_n) \in f$. In this case, as stated before, $s_{n-1} \in ms_1$, so $s_{n-2} \in ms_2$.

(2) $(s_{n-1}, s_n) \in \delta_e$. Since $(s_{n-1}, s_n) \in \delta_e \cap \delta_b$, from the fact the original program satisfies $spec$ from $S$, we conclude that $s_{n-1} \notin S$ (i.e., $s_{n-1} \notin S'$). In this case, all transitions starting from $s_{n-1}$ should be in $mt$. If this is not the case then this implies that there exists a state $s$ such that $(s_{n-1}, s)$ is not in $mt$ and we would have added it to $\delta'_p$ by Line 22, as $S_{n-1}$ is outside $S'$. Since all transitions from $s_{n-1}$ are in $mt$, $s_{n-1}$ is in $ms_1$ (by Line 7). Hence, $s_{n-2}$ is in $ms_2$. Continuing this argument further leads to the conclusion that $s_0 \in ms_2$. This is a contradiction, as we know $S'$ does not include any state in $ms_2$. Thus, any prefix of $p'[]_2\delta_e[]f$ satisfies $Sf$. Thus, *C2* holds.

**C3**: Any $(s_0, s_1) \in \delta_r$, is in $mt$. By construction, $\delta'_p$ does not have any transition in $mt$. Hence, *C3* holds. □

*Appendix A.2. Completeness*

Now, we focus on showing that Algorithm 2 is complete, that is, if there is a solution that satisfies the problem statement for adding failsafe fault-tolerance, Algorithm 2 finds one. The proof of completeness is based on the analysis of states that were removed from $S$. First, we note the following observation:

**Observation 4.** *For every state $s_0$ in $ms_2$ one of three cases below is true:*

1. $s_0 \in ms_1$
2. $\exists s_1 :: (s_0, s_1) \in \delta_e \wedge s_1 \in ms_1$
3. $\exists s_1 :: (s_0, s_1) \in \delta_e \cap \delta_b$

Now, for the rest of our discussion in this section, we assume that Algorithm 2 has declared failure for finding a failsafe fault-tolerant revision of program $p = \langle S_p, \delta_p \rangle$ with invariant $S$. Let $p'' = \langle S_p, \delta''_p \rangle$ with invariant $S''$ be any revision for program $p$. Also, let $f$ and $\delta_e$ be a set of fault transitions and a set of environment transitions for $p$ (i.e., $p''$), respectively. If there exists a computation prefix that reaches a state in $ms_2$, there exists a computation with that prefix that either reaches a state in $ms_1$, or executes a transition in $\delta_b$ (i.e., violates safety). Specifically, we have the following lemma.

**Lemma A2.** *If $\alpha$ is a prefix of a $p''[]_2\delta_e[]f$ computation and $\alpha = \langle s_i, \ldots, s_m \rangle$ such that $s_i \in ms_2$, or $\alpha = \langle s_0 \ldots, s_{i-1}, s_i \rangle$ such that $s_i \in ms_2$ and $(s_{i-1}, s_i) \in f \cup \delta''_p$, then, there exists a suffix $\beta$ such that $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation and*

- $\exists j : j \geq i : ((s_{j-1}, s_j) \in \delta_b)$, *or*
- $\exists j : j \geq i : (s_j \in ms_1)$.

**Proof.** From Observation 4, we know there are three cases for $s_i$. The first case is $s_i \in ms_1$. The theorem trivially holds for this case. In the second and the third case, there is an environment transition that either is a bad transition, or reaches a state in $ms_1$. This transition can occur even with the fairness assumption, as the computation has started in $s_j$, or we have reached $s_j$ with a fault or program transition. □

For states in $ms_1$ we have the following lemma that says in any given revision $p''$ for $p$, if there exists a computation prefix that reaches a state in $ms_1$, then there exists a computation with that prefix that executes a transition in $\delta_b$.

**Lemma A3.** *If $\alpha = \langle s_0, \ldots, s_{i-1}, s_i \rangle$ where $s_i \in ms_1$ is a prefix of a $p''[]_2\delta_e[]f$ computation, then, there exists a suffix $\beta = \langle s_{i+1}, s_{i+2}, \ldots \rangle$ such that $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation and $\exists j : j \geq i : (s_j, s_{j+1}) \in \delta_b$.*

**Proof.** We prove this inductively as we expand $ms_1$:
**Base case**: $ms_1 = \{s_0 | \exists s_1 :: (s_0, s_1) \in f \cap \delta_b\}$
Let $\beta$ be any $p''[]_2\delta_e[]f$ computation starting from $s_1$. Since fault transitions can execute in any state, $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation such that $(s_0, s_1) \in \delta_b$.
**Induction hypothesis**: Theorem holds for current $ms_1$.
**Induction step**: We show when we add a state to $ms_1$, the theorem still holds for $ms_1$. A state $s_0$ is added into $ms_1$ in three cases:

**Case 1** $\exists s_1 : s_1 \in ms_2 : (s_0, s_1) \in f$
In this case according to Lemma A2, a transition in $\delta_b$ may occur, or a state in $ms_1$ can be reached. Hence, according to the induction hypothesis, a transition in $\delta_b$ can occur in both cases.

**Case 2** $\exists s_1 :: (s_1 \in ms_1 \wedge (s_0, s_1) \in \delta_e) \wedge (\forall s_2 :: (s_0, s_2) \in mt)$
In this case, if according to fairness, $(s_0, s_1)$ can occur, state $s_1 \in ms_1$ can be reached by $(s_0, s_1)$ and according to the induction hypothesis, the theorem is proved. However, if $(s_0, s_1)$ cannot occur, some other transition in $\delta''_p \cup f$ should occur but we know such transition is in $mt$ and reaches a state in $ms_2$. Thus, according to Lemma A2, either a transition in $\delta_b$ can occur, or a state in $ms_1$ can be reached. Hence, according to the induction hypothesis, a transition in $\delta_b$ can occur in both cases.

**Case 3** $\exists s_1 :: ((s_0, s_1) \in \delta_e \cap \delta_b) \wedge (\forall s_2 :: (s_0, s_2) \in mt)$
In this case, if, according to fairness, $(s_0, s_1)$ can occur, by its occurrence a transition in $\delta_b$ has occurred. However, if $(s_0, s_1)$ cannot occur, some other transition in $\delta''_p \cup f$ should occur but we know such a transition is in $mt$ and reaches a state in $ms_2$. Thus, according to Lemma A2, either a transition in $\delta_b$ can occur or a state in $ms_1$ can be reached. Hence, according to the induction hypothesis, a transition in $\delta_b$ can occur in both cases. □

The last lemma that we use to prove the completeness of Algorithm 2 is Lemma A4. This lemma states that having any state in $ms_4$ in the invariant of any give revision $p''$ of program $p$ results in having new computation in the invariant that was not in the original program.

**Lemma A4.** *Let $p'' = \langle S_p, \delta''_p \rangle$ with invariant $S''$ be any revision for program $p = \langle S_p, \delta_p \rangle$ with invariant $S$ such that, $S'' \subseteq S'$ and $\delta''_p|S'' \subseteq \delta'_p|S'$ for $S'$ and $\delta'_p$ at the beginning of the loop on Lines 13–21. If $S''$ includes any state in $ms_4$ in any iteration of the loop on Lines 13–21, then there is a $p''[]_2\delta_e$ computation that starts from $S''$ that is not a $p''[]_2\delta_e$.*

**Proof.** Suppose there exists state $s_0 \in S'' \cap ms_4$. Then, there is a state, $s \in ms_3$ such that $(s_0, s) \in \delta_e$ and $(\exists s_1, s_2 :: (s, s_1) \in \delta_e \ \wedge (s, s_2) \in \delta_p\ ) \wedge (\nexists s_3 :: (s, s_3) \in \delta'_p)$. Since $\delta''_p|S'' \subseteq \delta'_p|S'$, $(\nexists s_3 :: (s, s_3) \in \delta''_p)$. Since $S''$ is closed in $p'' \cup \delta_e$, $s$ and $s_1$ are in $S''$, as well. Now, observe that $\langle s_0, s, s_1, \cdots \rangle$ is $p''[]_2\delta_e$ computation but it is not a $p[]_2\delta_e$ computation, as because of fairness, $(s, s_1) \in \delta_e$ cannot occur when there exists $(s, s_2) \in \delta_p$. □

**Theorem A2.** *Algorithm 2 is complete.*

**Proof.** Suppose program $p''$ with invariant $S''$ solves the addition problem. We show that at any point of Algorithm 2, $S''$ must always be a subset of $S'$. We prove this by looking at lines where we set $S'$. According to Lemma A2 and Lemma A3, $S''$ cannot have any transition in $ms_2$, because by starting from a state in $ms_2$, a computation may execute a bad transition. In addition, $S''$ must be closed in $\delta''_p$ and cannot have any deadlock state. Thus, $S'' \subseteq S'$ for $S'$ of Line 12. According to Lemma A4, $S''$

cannot include any state in $ms_4$, because otherwise there is a $p''[]_2\delta_e$ computation that is not $p[]_2\delta_e$ (contradiction to *C1*). Thus, $S'' \subseteq S'$ for $S'$ of Line 20. Thus, always we have $S'' \subseteq S'$. Our algorithm declares failure only when $S' = \emptyset$. Thus, if our algorithm does not find any solution, from $S'' \subseteq S'$, we have $S'' = \emptyset$ (contradiction to Definition 11). □

*Appendix A.3. Time Complexity*

**Theorem A3.** *Algorithm 2 is polynomial (in the state space of p)*

**Proof.** The proof follows from the fact that each statement in Algorithm 2 is executed in polynomial time and the number of iterations is also polynomial. □

## Appendix B. Soundness, Completeness, and Complexity of Algorithm 3

*Appendix B.1. Soundness*

First, we show that the program constructed by Algorithm 3 never reaches a state in $ms_1$. Specifically,

**Lemma A5.** *For any $p'[]_2\delta_e[]f$ computation $\langle s_0, s_1, \ldots\rangle$ where $s_0 \in S'$, there dose not exist $s_i$ such that $s_i$ is in $ms_1$.*

**Proof.** Consider a computation $\langle s_0, s_1, \ldots\rangle$ of $p'[]_2\delta_e[]f$ where $s_0 \in S'$. We proof by induction that for all $i \geq 0$, $s_i \notin ms_1$:
**Base case**: $i \in \{0, 1\}$
It is clear $s_0 \notin ms_1$, because $ms_1 \subseteq ms_2$ and $s_0 \in S'$ and by construction, we know $S' \cap ms_1 = \emptyset$ (see Line 33). Also, $s_1 \notin ms_1$, because otherwise $s_0 \in ms_2$ that is contradiction to $S' \cap ms_2 = \emptyset$.
**Induction hypothesis**: $\forall i : 0 \leq i \leq n : s_i \notin ms_1$
**Induction step**: $s_{n+1} \notin ms_1$
Suppose $s_{n+1} \in ms_1$. Then, $(s_n, s_{n+1}) \in mt$. By construction, the program does not have any transition in $mt$. Thus, we have two cases for $(s_n, s_{n+1})$:

**Case 1**: $(s_n, s_{n+1}) \in f$
In this case, $s_n \in ms_1$ (by Line 28) that is contradictory to the induction hypothesis.

**Case 2**: $(s_n, s_{n+1}) \in \delta_e$
In this case, $s_n \in ms_2$. If $n = 0$, then $s_0 \in ms_2$ that is contradiction to $S' \cap ms_2 = \emptyset$. If $n > 0$, then $(s_{n-1}, s_n) \in mt$. By construction, the program does not have any transition in $mt$. Thus, we have two cases for $(s_{n-1}, s_n)$

**Case 2.1**: $(s_{n-1}, s_n) \in f$:
In this case, $s_{n-1} \in ms_1$ that is contradictory to the induction hypothesis.

**Case 2.2**: $(s_{n-1}, s_n) \in \delta_e$:
In this case, as both $(s_{n-1}, s_n)$ and $(s_n, s_{n+1})$ are in $\delta_e$, according to the fairness assumption, there does not exist a transition $\delta'_p$ starting from $s_n$ and it means that $s_n$ is added to $ms_1$ by Line 28 which is in contradiction with the induction hypothesis. □

Since we never reach any state in $ms_1$ starting from $S'$ and since any state in $\neg(R \cup R_p)$ is in $ms_1$ by Line 23, we conclude we never reach $\neg(R \cup R_p)$. Thus, we have the following corollary:

**Corollary A1.** *$R \cup R_p$ in the last iteration of loop on Lines 6–43 is a $f$-span for the program resulted by Algorithm 3.*

The following lemma states that in every computation of the repaired program that starts from its invariant (i.e., $S'$), if the program reaches a state in $R$, in the rest of the computation it will reach $S'$. Specifically,

**Lemma A6.** *For every $\delta'_p[]_2\delta_e[]f$ computation $\langle s_0, s_1, \ldots\rangle$ such that $s_0 \in S'$, for $S'$ and $R$ in the last iteration of the loop on Lines 6–43, we have:*
$\forall s_i : s_i \in R : (\exists j : j \geq i : s_j \in S')$.

**Proof.** We prove this lemma by induction as we expand set $R$:
**Base case**: $R = S'$
The proof is trivial.
**Induction hypothesis**: Theorem holds for current $R$.
**Induction step**:
For any state $s_0$ that is added to $R$ we have

$$\Big(\nexists s_2 : s_2 \in (\neg(R \cup R_p) \cup ms_1) : (s_0, s_2) \in \delta_e\Big) \wedge$$
$$\Big(\big(\exists s_1 : s_1 \in (R \cup R_p) - ms_1 : (s_0, s_1) \in \delta_e\big) \wedge (\nexists s_2 :: (s_0, s_2) \in (\delta_e \cap \delta_b))\Big)$$
$$\vee s_0 \in R_p\Big)$$

Thus, any environment transition either reaches $R$ or $R_p$. In the second case, since we have reached a state in $R_p$ by an environment transition, and since from any state in $R_p$ there is a program transition to $R$ (cf. Line 17), based on the fairness assumption, the computation will reach $R$. Thus, in either case, we reach $R$. Since in the last iteration of loop on Lines 6–43, we do not change the set of transitions for states in $R$ of the previous iteration, based on the induction hypothesis, the lemma is proved. □

The following lemma states that when the repaired program starts at its invariant $S'$, if it reaches a state in $R_p - R$, in the rest of the its computation it will reach a state in $S'$. Specifically,

**Lemma A7.** *For every $\delta'_p[]_2\delta_e[]f$ computation $\langle s_0, s_1, \ldots\rangle$ such that $s_0 \in S'$, for $S'$, $R_p$ and $R$ in the last iteration of loop on Lines 6–43, we have:*
$\forall s_i : s_i \in R_p - R : (\exists j : j \geq i : s_j \in S')$.

**Proof.** Let $s_i \in (R_p - R)$. Any state that is not in $R$ is in $ms_2$. Thus, $(s_{i-1}, s_i) \in mt$. By construction, the program does not have any transition in $mt$. Thus, $(s_{i-1}, s_i) \in f \cup \delta_e$. If $(s_{i-1}, s_i) \in f$, then $s_{i-1} \in ms_1$ that is a contradiction to Lemma A5. Thus, $(s_{i-1}, s_i) \in \delta_e$. Since we have reached $s_i$ by an environment transition and $s_i \in R_p$, the computation will reach $R$ and according to Lemma A6, the computation will reach $S'$. □

Based on Lemmas A6 and A7, we have the following corollary that guarantees recovery to the invariant from $R \cup R_p$.:

**Corollary A2.** *For every $\delta'_p[]_2\delta_e[]f$ computation $\langle s_0, s_1, \ldots\rangle$ such that $s_0 \in S'$, for $S'$, $R_p$ and $R$ in the last iteration of loop on Lines 6–43, we have:*
$\forall s_i : s_i \in R \cup R_p : (\exists j : j \geq i : s_j \in S')$.

**Theorem A4.** *Algorithm 3 is sound.*

**Proof.** In order to show the soundness of our algorithm, we need to show that the three conditions of the problem statement are satisfied.

**C1**: Satisfaction of **C1** for Algorithm 3 is the same as that for Algorithm 2 stated in the proof of the Theorem A1.

**C2**: We need to show that $p'$ is a masking fault-tolerant revision for $p$. Thus, we need to show the constraints of Definition 16 are satisfied. From **C1**, $S' \subseteq S$, the assumption that $p[]_2\delta_e$ satisfies $spec$ from $S$, $S' \neq \emptyset$ and $S'$ is closed in $p' \cup \delta_e$, all constraints of Definition 10 are satisfied. Thus, $p'[]_2\delta_e$ 2-satisfies $spec$ from $S'$.

Let $spec = \langle Sf, Lv \rangle$. Consider prefix $c$ of $p'[]_2\delta_e[]f$ such that $c$ starts from a state in $S'$. If $c$ does not satisfy $Sf$, there exists a prefix of $c$, say $\langle s_0, s_1, \ldots, s_n \rangle$, such that it has a transition in $\delta_b$. W.l.o.g., let $\langle s_0, s_1, \cdots, s_n \rangle$ be the smallest such prefix. It follows that $(s_{n-1}, s_n) \in \delta_b$. Hence, $(s_{n-1}, s_n) \in mt$. By construction, $p'$ does not contain any transition in $mt$. Thus, $(s_{n-1}, s_n)$ is a transition of $f$ or $\delta_e$. If it is in $f$ then $s_{n-1} \in ms_1$ which is a contradiction to Lemma A5. If it is in $\delta_e$ then $s_{n-1} \in ms_2$ and $(s_{n-2}, s_{n-1}) \in mt$. Again, by construction, we know that $\delta'_p$ does not contain any transition in $mt$, so $(s_{n-2}, s_{n-1})$ is either in $f$ or $\delta_e$. If it is in $f$ then $s_{n-2} \in ms_1$ (contradiction to Lemma A5). If it is in $\delta_e$, as both $(s_{n-2}, s_{n-1})$ and $(s_{n-1}, s_n)$ are in $\delta_e$, according to the fairness assumption, there does not exist a transition of $\delta'_p$ starting from $s_{n-1}$ and it means that $s_{n-1} \in ms_1$, which is again a contradiction to Lemma A5. Thus, each prefix of $c$ does not have a transition in $\delta_b$. Therefore, any prefix of $p'[]_2\delta_e[]f$ satisfies $Sf$.

As $p'$ 2-satisfies $spec$ from $S'$ in environment $\delta_e$, any prefix of $p'[]_2\delta_e[]f$ 2-satisfies $Sf$ and according to Corollary A1 and Corollary A2, $p'$ is masking 2-$f$-tolerant to $spec$ from $S'$ in environment $\delta_e$ with fault-span $R \cup R_p$ for $R$ and $R_p$ in the last iteration of the loop on Lines 6–43.

**C3**: Any $(s_0, s_1) \in \delta_r$, is in $mt$. By construction, $p'$ does not have any transition in $mt$, so $C3$ holds. □

*Appendix B.2. Completeness*

Like the proof of the completeness of Algorithm 2, the proof of the completeness of Algorithm 3 is based on the analysis of states that are removed from S. For Algorithm 3, we focus on the iterations of the loop on Lines 6–43.

Similar to Observation 1, we have the following observation for Algorithm 3:

**Observation 5.** *In any given iteration $i$ of loop on 6–43, let $R$, $R_p$ and $ms_1$ be $R$, $R_p$ and $ms_1$ at the end of iteration $i$. Then, for any $s_0$ such that $s_0 \notin R$ and $\exists s_1 :: (s_0, s_1) \in \delta_e$, we have*
$(\exists s_2 : s_2 \in \neg(R \cup Rp) \cup ms_1 : (s_0, s_2) \in \delta_e) \vee$
$(\exists s_2 :: (s_0, s_2) \in \delta_e \cap \delta_b)$.

We also note the following observation:

**Observation 6.** *For any $s_0$ such that $s_0 \notin R$, either $s_0 \in \neg(R \cup R_p)$, or $\exists s_2 : s_2 \in \neg(R \cup Rp) \cup ms_1 : (s_0, s_2) \in \delta_e$.*

Lemmas A8, A9, A10 and A11, Corollary A3 and Theorem A5 provided in the following, hold for any given iteration $i$ of loop on Lines 6–43 assuming Algorithm 3 has declared failure. For these results, let $p'' = \langle S_p, \delta''_p \rangle$ with invariant $S''$ be any revision for program $p = \langle S_p, \delta_p \rangle$ with invariant $S$ such that $S'' \subseteq S'$, $\delta''_p \subseteq \delta'_p$ and $\delta''_p \cap mt = \emptyset$. Consider $S'$, $\delta'_p$ and $mt$ at the beginning of iteration $i$ and $ms_1$, $ms_2$, $R$ and $R_p$ at the end of the iteration. Also, let $f$ and $\delta_e$ be a set of fault transitions and a set of environment transitions for $p$ (i.e., $p''$), respectively.

The following lemma focuses on the situation where a given revision $p''$ reaches a state that our algorithm marks as $\neg(R \cup R_p)$.

**Lemma A8.** *For every $p''[]_2\delta_e[]f$ prefix $\alpha = \langle s_0, \ldots, s_i \rangle$ such that $s_i \in \neg(R \cup R_p)$, there exists a suffix $\beta$ such that $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation and*

- $(s_i, s_{i+1}) \in \delta_b$, *or*
- $s_{i+1} \in ms_1$, *or*
- $s_{i+1} \in \neg(R \cup R_p)$, *or*
- $s_{i+1} \in (R_p - R) \wedge s_{i+2} \in \neg(R \cup R_p)$.

**Proof.** There are two cases for $s_i$:

**Case 1**: $s_i$ is environment-enabled (see Definition 18) in prefix $\alpha = \langle s_0, s_1, \ldots, s_i \rangle$:
According to Observation 5, there exists $s \in \neg(R \cup R_p) \cup ms_1$ such that $(s_i, s) \in \delta_e$, $(s_i, s) \in \delta_e \cap \delta_b$. Any suffix that starts from $s$ proves the theorem.

**Case 2**: $s_i$ is not environment-enabled in prefix $\alpha = \langle s_0, s_1, \ldots, s_i \rangle$
The proof for this case is identical to the proof of Case 2 of Lemma 2. □

**Corollary A3.** *For every $p''[]_2\delta_e[]f$ prefix $\alpha = \langle s_0, \ldots, s_i \rangle$ such that $s_i \in \neg(R \cup R_p)$, there exists a suffix $\beta$ such that $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation and*

- $\exists j : j \geq i : (s_{j-1}, s_j) \in \delta_b$, *or*
- $\exists j : j \geq i : s_j \in ms_1$, *or*
- $\forall i : i \geq 0 : s_i \notin S'$.

The following lemma focuses on states that are marked as $ms_2$.

**Lemma A9.** *If $\alpha$ is a prefix of a $p''[]_2\delta_e[]f$ computation and $\alpha = \langle s_i, \ldots, s_m \rangle$ such that $s_i \in ms_2$, or $\alpha = \langle s_0 \ldots, s_{i-1}, s_i \rangle$ such that $s_i \in ms_2$ and $(s_{i-1}, s_i) \in f \cup \delta''_p$, then, there exists a suffix $\beta$ such that $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation and*

- $\exists j : j \geq i : ((s_{j-1}, s_j) \in \delta_b)$, *or*
- $\exists j : j \geq i : (s_j \in ms_1)$, *or*
- $\forall i : i \geq 0 : s_i \notin S'$.

**Proof.** We prove this lemma by looking at lines where we expand $ms2$:

**Line 4**: $ms_2 = ms_1 \cup \{s_0 | \exists s_1 : (s_0, s_1) \in \delta_e \cap \delta_b\}$
In this case, we add $s_0$ to $ms_2$ if either it is in $ms_1$, or it has a $(s_0, s_1)$ transition that is in $\delta_b$. If $s_0 \in ms_1$, any computation starting from $s_0$ proves the theorem. Otherwise, any computation $\langle s_0, s_1, \ldots \rangle$ proves the theorem.

**Line 24**: $ms_2 = ms_2 \cup \neg R$:
According to Observation 6, there is a suffix (possibly $\langle\rangle$) that reaches $\neg(R \cup R_p)$. According to Corollary A3 there is a computation that either reaches $ms_1$, or never reaches $S'$.

**Line 29**: $ms_2 = ms_2 \cup ms_1 \cup \{s_0 | \exists s_1 : s_1 \in ms_1 : (s_0, s_1) \in \delta_e)\}$

In this case, we add state $s_0$ to $ms_2$, if $s_0$ is in $ms_1$ or can reach state $s_1 \in ms_1$ with an environment transition. If $s_0 \in ms_1$, any computation starting from $s_0$ proves the theorem. Otherwise, any computation $\langle s_0, s_1, \ldots \rangle$ proves the theorem. Note that, since we have started the computation from $s_0$, or we have reached $s_0$ with a fault or program transition, even with the fairness assumption, the environment transition $(s_0, s_1)$ can execute. □

The following lemma states that reaching any state in $ms_1$ will result in bad consequences that can be either executing a bad transition or never recovering to the invariant. Specifically,

**Lemma A10.** *If $\alpha = \langle s_0, \ldots, s_{i-1}, s_i \rangle$ where $s_i \in ms_1$ is a prefix of a $p''[]_2\delta_e[]f$ computation, then, there exists a suffix $\beta = \langle s_{i+1}, s_{i+2}, \ldots \rangle$ such that $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation and*

- $\exists j : j \geq i : (s_j, s_{j+1}) \in \delta_b$, *or*

- $\forall j : j \geq 0 : s_j \notin S'$.

**Proof.** We prove this theorem inductively based on where we expand $ms_1$:
**Base Case**: $ms_1 = \{s_0 | (s_0, s_1) \in f \cap \delta_b\}$
Let $\beta$ be any $p''[]_2\delta_e[]f$ computation starting from $s_1$. Since fault transitions can execute in any state, $\alpha\beta$ is a $p''[]_2\delta_e[]f$ computation such that $(s_0, s_1) \in \delta_b$.
**Induction hypothesis**: Theorem holds for current $ms_1$.
**Induction step**: We look at lines where we add a state to $s_0$:

**Line 23**: $ms_1 = ms_1 \cup \neg(R \cup R_p)$
According to Corollary A3, there is a suffix that either runs a transition in $\delta_b$, never reaches $S'$, or reaches a state in $ms_1$. Thus, the theorem is proved by the induction hypothesis.

**Line 28**: $ms_1 = ms_1 \cup \{s_0 | \exists s_1 : s_1 \in ms_2 : (s_0, s_1) \in f\} \cup \{s_0 | (\exists s_1 : s_1 \in ms_1 : (s_0, s_1) \in \delta_e) \vee (s_0, s_1) \in (\delta_e \cap \delta_b)) \wedge (\nexists s_2 :: (s_0, s_2) \in \delta'_p)\}$
We add state $s_0$ to $ms_1$ in three cases in this line:

**Case 1** $\exists s_1 : s_1 \in ms_2 : (s_0, s_1) \in f$
In this case according to Lemma A9, a transition in $\delta_b$ may occur, or there is a suffix that never reach $S'$, or a state in $ms_1$ can be reached. Thus, according to the induction hypothesis, the theorem is proved.

**Case 2** $\exists s_1 :: (s_1 \in ms_1 \wedge (s_0, s_1) \in \delta_e) \wedge (\nexists s_2 :: (s_0, s_2) \in \delta'_p)$
In this case, if according to fairness, $(s_0, s_1)$ can occur, state $s_1 \in ms_1$ can be reached by $(s_0, s_1)$ and according to the induction hypothesis, the theorem is proved. However, if $(s_0, s_1)$ cannot occur, some other transition $t \in \delta''_p \cup f$ occurs. Since $\exists s_1 : s_1 \in ms_1 : (s_0, s_1) \in \delta_e$, we know that $s_0 \notin R$. By construction, $\delta'_p$ contains any transition from states $\neg R$ to $R$. Since $\nexists s_2 :: (s_0, s_2) \in \delta'_p$, we conclude $t$ goes to a state in $\neg R$ (i.e., $ms_2$). Thus, according to Lemma A9 either a transition in $\delta_b$ can occur, or there is a suffix that never reaches $S'$, or a state in $ms_1$ can be reached. Thus, according to the induction hypothesis, the theorem is proved.

**Case 3** $\exists s_1 :: ((s_0, s_1) \in \delta_e \cap \delta_b) \wedge (\nexists s_2 :: (s_0, s_2) \in \delta'_p)$
In this case, if according to fairness, $(s_0, s_1)$ can occur, by its occurrence a transition in $\delta_b$ has occurred. However, if $(s_0, s_1)$ cannot occur, some other transition in $\delta''_p$ should occur. Since $\exists s_1 :: ((s_0, s_1) \in \delta_e \cap \delta_b)$, $s_0 \notin R$. Our algorithm add any possible program transition that is not $mt$ and goes to a state in $R$ to $s_0$ in Line 17. Since there is no such transition, any transition in $\delta''_p$ either is in $mt$ or goes to $\neg R$. In either case, we have a computation that reaches a state in $ms_2$ starting from $s_0$. Thus, according to Lemma A9, either a transition in $\delta_b$ can occur, or there is suffix that never reaches $S'$, or a state in $ms_1$ can be reached. Thus, according to the induction hypothesis, the theorem is proved.
□

Like Lemma A4 for Algorithm 2, we have following lemma for $ms_4$ for Algorithm 3.

**Lemma A11.** *If $S''$ includes any state in $ms_4$ in any iteration of the loop on Lines 13–21, then there is a $p''[]_2\delta_e$ computation that starts from $S''$ that is not a $p''[]_2\delta_e$.*

**Proof.** The proof of this lemma is very similar to that of Lemma A4. □

**Theorem A5.** *Algorithm 3 is complete.*

**Proof.** Suppose program $p''$ and invariant $S''$ solve transformation problem. We show that at any point of Algorithm 3, $S''$ must always be a subset of $S'$. We prove this by looking at lines where we set $S'$.

In the first iteration of the loop on Lines 6–43, $S' = S$. According to constraint **C1** of the problem definition in Section 5.1, $S'' \subseteq S$. Thus, $S'' \subseteq S'$ for the $S'$ at the beginning of the first iteration of the loop on Lines 6–43. According to Lemmas A9 and A10, $S''$ cannot have any transition in $ms_2$ in first iteration of the loop on Line 6–43, because by starting from a state in $ms_2$, a computation may execute a bad transition, or reach a state outside $S''$ from which there is a computation that never reaches $S'$ (i.e., never reaches $S''$). In addition, $S''$ must be closed in $\delta''_p$ and cannot have any deadlock state. Thus, $S'' \subseteq S'$ for $S'$ at Line 33.

According to Lemma A11, $S''$ cannot include any state in $ms_4$ in the first iteration of the loop of Lines 6–43, because otherwise there is a $p''[]_2\delta_e$ computation that is not $p[]_2\delta_e$ (contradiction to **C1**). Thus, $S'' \subseteq S'$ for $S'$ at the end of the first iteration of the loop on Lines 6–43.

With the induction, we conclude that $S''$ cannot include any states in $ms_2$ or $ms_4$ in next iterations of the loop of Lines 6–43. Thus, always we have $S'' \subseteq S'$. Our algorithm declares failure only when $S' = \emptyset$. Thus, if our algorithm does not find any solution, from $S'' \subseteq S'$, we have $S'' = \emptyset$ (contradiction to Definition 11). □

*Appendix B.3. Time Complexity*

**Theorem A6.** *Algorithm 3 is polynomial (in the state space of p)*

**Proof.** t The proof follows from the fact that each statement in Algorithm 3 is executed in polynomial time and the number of iterations is also polynomial. □

## References


1. Bonakdarpour, B.; Kulkarni, S.S.; Abujarad, F. Symbolic Synthesis of Masking Fault-tolerant Programs. *Springer J. Distrib. Comput.* **2012**, *25*, 83–108. [CrossRef]
2. Bonakdarpour, B.; Kulkarni, S.S. Active Stabilization. In Proceedings of the Stabilization, Safety, and Security of Distributed Systems, 13th International Symposium, SSS 2011, Grenoble, France, 10–12 October 2011; Volume 6976, pp. 77–91.
3. Roohitavaf, M.; Kulkarni, S. Stabilization and Fault-tolerance in Presence of Unchangeable Environment Actions. In Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, 4–7 January 2016; ACM: New York, NY, USA, 2016; pp. 19:1–19:10. [CrossRef]
4. JavaRepair. Available online: https://github.com/roohitavaf/JavaRepair (accessed on 1 July 2019).
5. Alpern, B.; Schneider, F.B. Defining liveness. *Inf. Process. Lett.* **1985**, *21*, 181–185. [CrossRef]
6. Arora, A.; Gouda, M.G. Closure and Convergence: A Foundation of Fault-Tolerant Computing. *IEEE Trans. Softw. Eng.* **1993**, *19*, 1015–1027. [CrossRef]
7. Dijkstra, E.W. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM* **1974**, *17*, 643–644. [CrossRef]
8. Dolev, S. *Self-Stabilization*; MIT Press: Cambridge, MA, USA, 2000.
9. Kundur, D.; Feng, X.; Liu, S.; Zourntos, T.; Butler-Purry, K. Towards a Framework for Cyber Attack Impact Analysis of the Electric Smart Grid. In Proceedings of the First IEEE International Conference on Smart Grid Communications, Gaithersburg, MD, USA, 4–6 October 2010; pp. 244–249.
10. Roohitavaf, M.; Kulkarni, S. Collaborative Stabilization. In Proceedings of the 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS), Budapest, Hungary, 26–29 September 2016; pp. 259–268.
11. Buccafurri, F.; Eiter, T.; Gottlob, G.; Leone, N. Enhancing Model Checking in Verification by AI Techniques. *Artif. Intell.* **1999**, *112*, 57–104. [CrossRef]
12. Chatzieleftheriou, G.; Bonakdarpour, B.; Smolka, S.A.; Katsaros, P. Abstract Model Repair. In Proceedings of the NASA Formal Methods Symposium (NFM), Norfolk, VA, USA, 3–5 April 2012.
13. Bonakdarpour, B.; Ebnenasir, A.; Kulkarni, S.S. Complexity Results in Revising UNITY Programs. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **2009**, *4*, 5. [CrossRef]
14. Chandy, K.M.; Misra, J. *Parallel Program Design: A Foundation*; Addison-Wesley Longman Publishing Co. Inc.: Boston, MA, USA, 1988.

15. Samanta, R.; Deshmukh, J.V.; Emerson, E.A. Automatic Generation of Local Repairs for Boolean Programs. In Proceedings of the Formal Methods in Computer-Aided Design (FMCAD), Portland, OR, USA, 17–20 November 2008; pp. 1–10.
16. Roohitavaf, M.; Kulkarni, S.S. Automatic Addition of Conflicting Properties. In Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems, Lyon, France, 7–10 November 2016; pp. 310–326.
17. Ramadge, P.J.; Wonham, W.M. The control of discrete event systems. *Proc. IEEE* **1989**, *77*, 81–98. [CrossRef]
18. Girault, A.; Rutten, É. Automating the addition of fault tolerance with discrete controller synthesis. *Form. Methods Syst. Des. (FMSD)* **2009**, *35*, 190–225. [CrossRef]
19. Cho, K.H.; Lim, J.T. Synthesis of Fault-Tolerant Supervisor for Automated Manufacturing Systems: A Case Study on Photolithography Process. *IEEE Trans. Robot. Autom.* **1998**, *14*, 348–351.
20. Chen, J.; Roohitavaf, M.; Kulkarni, S.S. Ensuring Average Recovery with Adversarial Scheduler. In Proceedings of the LIPIcs-Leibniz International Proceedings in Informatics, Rennes, France, 14–17 December 2015.
21. Roohitavaf, M.; Lin, Y.; Kulkarni, S.S. Lazy repair for addition of fault-tolerance to distributed programs. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium, Chicago, IL, USA, 23–27 May 2016; pp. 1071–1080.
22. Pnueli, A.; Rosner, R. On the synthesis of a reactive module. In Proceedings of the Principles of Programming Languages (POPL), Austin, Texas, USA, 11–13 January 1989; pp. 179–190.
23. Pnueli, A.; Rosner, R. On The Synthesis of An Asynchronous Reactive Module. In *International Colloquium on Automata, Languages, and Programming*; Springer: Berlin/Heidelberg, Germany, 1989; pp. 652–671.
24. Jobstmann, B.; Griesmayer, A.; Bloem, R. Program Repair as a Game. In Proceedings of the Conference on Computer Aided Verification (CAV), Scotland, UK, 6–10 July 2005; pp. 226–238.
25. Bonakdarpour, B.; Lin, Y.; Kulkarni, S.S. Automated Addition of Fault Recovery to Cyber-physical Component-based Models. In Proceedings of the ACM International Conference on Embedded Software (EMSOFT), New York, NY, USA, 9–14 October 2011; pp. 127–136.
26. Hajisheykhi, R.; Roohitavaf, M.; Kulkarni, S.S. Auditable restoration of distributed programs. In Proceedings of the 2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS), Montreal, QC, Canada, 28 September–1 October 2015; pp. 37–46. [CrossRef]
27. Hajisheykhi, R.; Roohitavaf, M.; Kulkarni, S.S. Bounded auditable restoration of distributed systems. *IEEE Trans. Comput.* **2016**, *66*, 240–255. [CrossRef]
28. Bonakdarpour, B.; Bozga, M.; Jaber, M.; Quilbeuf, J.; Sifakis, J. A framework for automated distributed implementation of component-based models. *Distrib. Comput.* **2012**, *25*, 383–409.