Stabilization and Fault-Tolerance in Presence of Unchangeable Environment Actions

Mohammad Roohitavaf
Computer Science and Engineering Department
Michigan State University
East Lansing, Michigan
48824, USA
roohitav@cse.msu.edu

Sandeep Kulkarni
Computer Science and Engineering Department
Michigan State University
East Lansing, Michigan
48824, USA
sandeep@cse.msu.edu

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 algorithms to declare failure to add fault-tolerance.

We present algorithms for adding stabilizing and fault-tolerance. Since previous approaches for adding stabilizing and fault-tolerance to (disruptive and eventually terminating) faults cannot be extended for environment actions, we develop new algorithms that are sound, complete and in P (in the state space).

CCS Concepts
• Software and its engineering → Formal methods;
• Software verification and validation;
• Computer systems organization → Dependable and fault-tolerant systems and networks;
• Embedded and cyber-physical systems;

Keywords
Stabilization, Fault-tolerance, Program synthesis, Addition of fault-tolerance

1. INTRODUCTION
In this paper, we focus on the problem of model repair for the purpose of making the model stabilizing or fault-tolerant. 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 to 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 that actions can be 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 (CPSs) where modifying physical components may be very expensive or even impossible.

The environment actions differ from fault actions considered in [5]. Fault actions are assumed to be temporary in nature, and all the previously proposed algorithms to add fault-tolerance in [5], work only with this important assumption that faults finally stop occurring. However, unlike fault actions, environment actions can keep occurring. Environment actions also differ from adversary actions considered in [4] 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, the 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 them while overcoming their disruption. To give an intuition of the role of the environment and the difference between program, environment, and fault actions, next, we present the following example.

An intuitive example to illustrate the role of environment. This 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 sake of presentation, assume that pressure is below 4 in normal states. 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 at the vent pipe),
Overpressure valve

Pressure Cooker Environment Action

Figure 1: An intuitive example to illustrate the role of environment actions. For sake of readability, only transitions relevant to the discussion are shown in the figure.

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 pressure is a by $s_a$ when the vent is working, and by state $fs_a$ when the vent has failed.

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 the role of the environment differs from that of fault actions and program actions. In turn, this prevents us from using existing approaches such as [5]. 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. In 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 be 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 transition from $s_6$ (respectively $fs_6$) to where 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 environment actions remain unchanged. We cannot introduce new environment transitions and we cannot remove existing environment transitions. This is what we mean by 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, i.e., if it reaches a state where only environment actions can execute then one of them does execute. (Note that this requirement cannot be expected of faults.) 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 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.)

**Goal of the paper.** Based on the above example, 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.

We also note that the results in [5] do not model environment actions. Using the framework in [5] for the above example would require one to treat the environment actions to be fault actions. And, as discussed above, this leads to an unacceptable result.

**Contributions of the paper.** The main results of this work are as follows:

- We present an algorithms for addition of stabilization to an existing program. This algorithm is designed for the case where the program is provided with minimal fairness (where the program is given a chance to execute at least once between any two environment actions). This algorithm is sound and complete, i.e., 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 an algorithm for addition of fault-tolerance. In this paper, we focus on failsafe fault-tolerance. This algorithm is also sound and complete.

- We show that the complexity of all algorithms presented in this paper is polynomial (in the state space of the program).

- We have extended the algorithm for adding stabilization to deal with the case where the program is provided additional fairness. This algorithm is especially
applicable when adding stabilization with minimal fairness is impossible. We have also developed an algorithm for addition of masking fault-tolerance. Due to the reason of space, we refer the reader to [20] for these algorithms.

Organization of the paper. This paper is organized as follows: in Section 2 we provide the definitions of a program design, specifications, faults, fault-tolerance, and safe stabilization. In Section 3 we define the problem of adding safe stabilization, and propose an algorithm to solve that problem for the case of minimal fairness. In Section 4, as a case study, we illustrate how adding stabilization algorithm can be used for the controller of a smart grid. In Section 5, we define the problem of adding fault-tolerance, and propose the algorithm to add failable fault-tolerance. In Section 6 we show how our proposed algorithms can be extended to solve related problems. In Section 7, we discuss related work. In section 8, we discuss application of our algorithms for cyber-physical and distributed systems. Finally, we make concluding remarks in Section 9.

2. PRELIMINARIES

In this section, we define the notion of programs, specifications, faults, and fault-tolerance. We define programs in terms of their states and transitions. The definitions of specification is based on that by Alpern and Schneider [1]. And, the definitions of faults and fault-tolerance are adapted from those by Arora and Gouda [2].

2.1 Program Design Model

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

The environment in which the program executes also changes the state of the program. Instead of modeling this 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 where models it as a subset of $S_p \times S_p$. Thus,

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

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

**Definition 4 (Projection).** The projection of program $p$ on state predicate $S$, denoted as $p[S]$, is the program $(S_p, \{(s_0, s_1) : (s_0, s_1) \in \delta_p \land s_0, s_1 \in S\})$. In other words, $p[S]$ consists of transitions of $p$ that start in $S$ and end in $S$. We denote the set of transitions of $p[S]$ by $\delta_p[S]$.

We 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 $\delta_p \cup \delta_e$, we add the self-loop transition $(s_0, s_0)$ to $\delta_e$. 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.

**Definition 5 (p][\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 $(s_0, s_1, s_2, ...)$ is a $p][\delta_e$ 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
- $\forall i : i \geq 0 : ((s_i, s_{i+1}) \in \delta_e) \Rightarrow (\forall j : 0 < l < i \Rightarrow (\exists s'_l : (s_i, s'_l) \in \delta_p) \Rightarrow (s_i, s_{i+1}) \in \delta_p))$.

Note that the above definition requires that in every step, either a program transition or an environment transition is executed. Moreover, after the environment transition executes, the program is given a chance to execute in the next $k-1$ steps. However, in any state that no program transition is available, an environment transition can execute.

**Definition 6 (Closure).** A state predicate $S$ is closed in a set of transitions $\delta$ iff $(\forall(s_0, s_1) : (s_0, s_1) \in \delta : (s_0 \in S \Rightarrow s_1 \in S))$.

2.2 Specification

Following Alpern and Schneider [1], we let the specification of program to consist of a safety specification and a liveness specification.

**Definition 7 (Safety).** The safety specification is specified in terms of a set of transitions, $\delta_0$, that the program is not allowed to execute. Thus, a sequence $\sigma = \langle s_0, s_1, ... \rangle$ refines the safety specification $\delta_0$ iff $\forall j : 0 < j < \text{length}(\sigma) : (s_j, s_{j+1}) \notin \delta_0$.

**Definition 8 (Liveness).** The liveness specification is specified in terms of a leads-to property $(L \sim T)$ to denote, where both $L$ and $T$ are state predicates. Thus, a sequence $\sigma = \langle s_0, s_1, ... \rangle$ refines the liveness specification iff $\forall j : L$ is true in $s_j : (\exists k : j < k < \text{length}(\sigma) : T$ is true in $s_k$).

**Definition 9 (Specification).** A specification, is a tuple $(S_f, L_v)$, where $S_f$ is a safety specification and $L_v$ is a liveness specification. A sequence $\sigma$ refines spec iff it refines $S_f$ and $L_v$.

**Definition 10 (Refines).** $p][\delta_e$ refines spec from $S$ iff the following conditions hold:

- $S$ is closed in $\delta_e \cup \delta_e$, and
- Every computation of $p]][\delta_e$ that starts from a state in $S$ refines spec.

We note that from the above definition, it follows that starting from a state in $S$, execution of either a program action or an environment action 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).

**Definition 11 (Invariant).** If $p$ refines spec from $S$ and $S \neq \phi$, we say that $S$ is an invariant of $p$ for spec.

2.3 Faults and Fault-Tolerance

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

**Definition 13 (p][\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 $(s_0, s_1, s_2, ...)$ is a $p][\delta_e[f$ computation iff
3. Addition of Safe Stabilization

In this section, we present our algorithm for adding safe stabilization to an existing program. In Section 3.1, we identify the problem statement. In Section 3.2, we present our algorithm for the case where the parameter $k$ (that identifies the fairness between program and environment actions) is set to 2. Due to reasons of space, the algorithm for arbitrary value of $k$ is presented in [20].

3.1 Problem Definition

The problem for adding safe stabilization begins with a program $p$, its invariant $S$, and a safety specification $\delta_b$ that identifies the set of bad transitions. The goal is to add stabilization so that starting from an arbitrary state, the program recovers to $S$. Moreover, we want to ensure that during recovery the program does not execute any transition in $\delta_b$. Also, we want to make sure that the execution of environment actions cannot prevent recovery to $S$. Thus, the problem statement is as follows:

Given program $p$ with state space $S_p$ and transitions $\delta_e$, state predicate $S$, set of bad transitions $\delta_b$, environment $\delta_e$, and $k > 1$, identify $p'$ with state space $S_p$ such that:

- $p'[S = p]S$
- $p'[\delta_e]$ is $\delta_b$-safe stabilizing for invariant $S$

3.2 Algorithm to Add Safe Stabilization

In this section, we present an algorithm for the problem of addition of stabilization defined in the Section 3.1. The algorithm proposed here adds stabilization for $k = 2$. When $k = 2$, the environment transition 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. Observe that if $\delta_b \cap \delta_e$ is nonempty then adding stabilization is impossible. This is due to the fact that if the program starts in a state where such a transition can execute then it can immediately violate safety. Hence, this algorithm (but not the algorithms for adding failsafe fault-tolerance) assumes that $\delta_b \cap \delta_e = \phi$.

The algorithm 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 should be equal to the original program. Therefore, in the first line, we set $\delta'_p$ to $\delta_p[S]$. State predicate $R$ is the set of states such that every computation starting from $R$ has a state in $S$. Initially (Line 2) $R$ is initialized to $S$. In each iteration, state predicate $R_p$ is the set of states that can reach a state in $R$ using a safe program transition, i.e., a transition not in $\delta_b$. In Line 9 we add such program transitions to $\delta'_p$.

Algorithm 1 Addition of safe stabilization

| Input: $S_p, \delta_p, \delta_e, S,$ and $\delta_b$ |
| Output: $\delta'_p$ or Not-Possible |

1: $\delta'_p := (\delta_p)[S]$;
2: $R = S$;
3: $R_p = \emptyset$;
4: repeat
5: $R' = R$;
6: $R'_p = \{s_0|s_0 \notin (R \cup R_p) \land \exists s_1 : s_1 \in R : (s_0, s_1) \notin \delta_b\}$;
7: $R_p = R_p \cup R'_p$;
8: for each $s_0 \in R_p$ do
9: $\delta'_p = \delta'_p \cup \{s_0, s_1\}$, $s_0 \notin \delta_b \land s_1 \in R$;
10: end for
11: if $\exists s_0 \notin R R_p$ then
12: $R = R \cup s_0$;
13: end for
14: until $(R' = R)$;
15: if $\exists s_0 \notin R$ then return Not-Possible;
16: else
17: return $\delta'_p$;
18: 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 $s_0$ and going to state outside $R \cup R_p$. In addition to this condition, there should 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 the last 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 2 to illustrate Algorithm 1. Figure 2 depicts the status of the state space in a hypothetical $i^{th}$ iteration of loop on Lines 4-14. In this iteration state $A$ is added to $R$. 
If $s_1$ is in $R_p$, there is a program transition from $s_1$ to a state in $R$. As $(s_0, s_1) \in \delta_c$, because of fairness assumption, the program can occur, and reach $R$. Thus, every computation starting from $s_0$ has a state in $R$ (in the previous iteration). Since we do not change the set of transitions of any state in $R$ or $R_p$ in the previous iteration, the set of computations of any state in $R$ is unchanged. Thus, every computation starting from $s_0$ has a state in $S$.

\textbf{Case 2} \ \exists s_2 : s_2 \in \neg(R \cup R_p) : (s_0, s_2) \in \delta_c \land s_0 \in R_p

Since there is no $s_2$ in $\neg(R \cup R_p)$ such that $(s_0, s_2) \in \delta_c$, for every $(s_0, s_1) \in \delta_c$, $s_1$ is either in $R$ or $R_p$. In addition, we know that there is at least one state $s_1$ in $R \cup R_p$ such that $(s_0, s_1) \in \delta_c$. In any computation of $p[\alpha][k]s_c$ starting from $s_0$ if $(s_0, s_1) \in \delta_p$ then $s_1 \in R$. If $(s_0, s_1) \in \delta_c$, then $s_1 \in R \cup R_p$. If $s_1$ is in $R_p$, there is a program transition from $s_1$ to a state in $R$. As $(s_0, s_1) \in \delta_c$, because of fairness assumption program can reach $R$. Thus, every computation starting from $s_0$ has a state in $R$. As explained in the previous case, we do not change the set of computations starting from any state in $R$. Hence, every computation starting from $s_0$ has a state in $S$.

\textbf{THEOREM 1. Algorithm 1 is sound.}

\textbf{Proof.} 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 $S$, so $p[\alpha][k]s_c$ always 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.

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

\textbf{Observation 2.} For any $s_0$ such that $s_0 \notin R$ and $s_1 \in (R \cup R_p)$, we have $\exists s_2 : s_2 \notin R \cup R_p$.

\textbf{Lemma 2.} Let $\delta''_p$ be any program such that $\delta''_p \cap \delta_c = \emptyset$. Let $s_j$ be any state in $\neg(R \cup R_p)$. Then, either $s_j$ is a deadlock state in $\delta''_p \cup \delta_c$, or for every $p[\alpha][k]s''_c$ such that $\alpha \delta$ is a computation, one of two conditions below is correct:

1. $s_j \in (R \cup R_p)$
2. $s_{j+1} \in R_p \land s_{j+2} \in (R \cup R_p)$

\textbf{Proof.} There are two cases for $s_j$:

\textbf{Case 1} If $s_j$ is environment-enabled

Based on the Observation 2 there should exist $s'' \in \neg(R \cup R_p)$ such that $s_j \in \neg(R \cup R_p)$, $s_{j+1} \in (R \cup R_p)$. 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 1, we have \( \exists s_2 : s_2 \in \neg (R \cup R_p) : (s_{j+1}, s_2) \in \delta_\phi \). As \( (s_j, s_{j+1}) \in \delta_p \), even with fairness \( (s_{j+1}, s_2) \) can occur. Therefore we set \( s_{j+2} = s_2 \), i.e., \( s_{j+2} \in \neg (R \cup R_p) \).

\[ \neg \exists \]

**Corollary 1.** Let \( \delta_p^{\text{ref}} \) be any program such that \( \delta_p^{\text{ref}} \cap \delta_\phi = \phi \). Let \( s_j \) be any state in \( \neg (R \cup R_p) \). Then for every \( p'' \ | \ |_{\delta_\phi} \) prefix \( \alpha = [\ldots, s_{j-1}, s_j] \), there exists suffix \( \beta = (s_{j+1}, s_{j+2}, \ldots) \), such that \( \alpha \beta \) is a \( p'' \ | \ |_{\delta_\phi} \) computation, and \( \forall i : t : s_i \in \neg R \) (i.e., \( \neg S \)).

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

**Proof.** Algorithm 1 returns Not-possible only when at the end of loop there exists a state \( s_0 \) such that \( s_0 \notin R \). When \( s_0 \notin R \), according to Observation 1 we have two cases as follows:

Case 1 \( \exists s_2 : s_2 \in \neg (R \cup R_p) : (s_0, s_2) \in \delta_\phi \)
As there exists an environment action to state \( s_2 \) in \( \neg (R \cup R_p) \), starting from \( s_0 \) there is a computation that next step is in \( \neg (R \cup R_p) \). Note that, when a computation starts from \( s_0 \), even with fairness assumption \( (s_0, s_2) \in \delta_\phi \) can occur. Based on Corollary 1, for every \( \delta_p^{\text{ref}} \) such that \( \delta_p^{\text{ref}} \cap \delta_\phi = \phi \), starting from \( s_0 \), there is a computation such that every state is in \( \neg R \).

Case 2 \( \not\exists s_1 : s_1 \in (R \cup R_p) : (s_0, s_1) \in \delta_\phi \)
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 \( \delta_p^{\text{ref}} \) such that \( \delta_p^{\text{ref}} \cap \delta_\phi = \phi \), there is a computation starting from \( s_0 \) such that all states are outside \( R \) (i.e, outside \( s \)). Thus, it is impossible to have any stabilizing revision for the program.

**Theorem 3.** Algorithm 1 is polynomial (in the 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 are 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 is used to add safe stabilization to a controller program of a smart grid. We consider an abstract version of the smart grid described in [16] (see Figure 3). 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 shedding is used if the load is too high (respectively, generating capacity is enough).

The control center is shown by a dashed circle in Figure 3. It can read the values of the sensors and turn on/off switches connected to the loads. The program of the control center should control switches in a manner that all the conditions below are satisfied:

1. Both switches should 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 3 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 \( [0, \text{max}] \). And, 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:

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

**Observation 3.** For any value of \( V_1, V_2, \) and \( V_G \), there exists an assignment to \( w_1 \) and \( w_2 \) such that the resulting state is in \( S \).

\[ ^1 \text{We need to add } 0 \leq V_1, V_2, V_G \leq \text{max} \text{ to all conditions. For brevity, we keep these implicit.} \]
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, or leave the invariant. Thus, set of environment transitions, $\delta_e$ is equal to $\{(s_0, s_1) | \{w_i(s_0) = w_i(s_1)\} \land (w_2(s_0) = w_2(s_1)) \land (V_G(s_0) = V_G(s_1)) \land (\bigwedge_{i=1}^d I_i(s_0) \Rightarrow \bigwedge_{i=1}^d I_i(s_1))\}$, 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 bad transitions, $\delta_b$ for this program is equal to $\{(s_0, s_1) | V_G(s_0) \neq V_G(s_1) \lor V_i(s_0) \neq V_i(s_1) \lor V_2(s_0) \neq V_2(s_1) \lor (w_i(s_0) \neq w_1(s_1) \land w_2(s_0) \neq w_2(s_1))\}$.

For the sake of presentation and to illustrate the role of $k$, we also assume 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 bad transitions. We call the set of bad transitions for this case $\delta_{b_2}$ and it is equal to $\{(s_0, s_1) | V_G(s_0) \neq V_G(s_1) \lor V_i(s_0) \neq V_i(s_1) \lor V_2(s_0) \neq V_2(s_1) \lor (w_1(s_0) \neq w_1(s_1) \land w_2(s_0) \neq w_2(s_1))\}$.

### 4.2 Adding 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 bad transitions, $\delta_b$ and $\delta_{b_2}$.

#### 4.2.1 Adding Stabilization for $\delta_b$

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, following set of transitions are added to $\delta_b$ by Line 9:

$$\{(s_0, s_1) | V_i(s_0) = V_i(s_1) \land V_2(s_0) = V_2(s_1) \land V_G(s_0) = V_G(s_1) \land s_0 \notin \bigcup_{i=1}^d I_i \land s_1 \in \bigcup_{i=1}^d I_i\}$$

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 Line 4-14 terminates. Since there is no state in $\neg R$, the algorithm returns $\delta_b^\prime$ as the transition of the resulting $\delta_b$-safe stabilizing program for $S$.

#### 4.2.2 Adding Stabilization for $\delta_{b_2}$

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 $\delta_{b_2}$, it cannot change the status of both switches. Therefore, state predicate $R_p$ is the union of state predicates $R_{p_1}$ to $R_{p_0}$ as follows ($\oplus$ denotes the xor operation):

$$R_{p_1} = (V_1 + V_2 \leq V_G) \land (w_1 \oplus w_2)$$

$$R_{p_0} = (V_1 \leq V_G \land V_2 > V_G) \land (w_1 \oplus w_2)$$

$$R_{p_1} = (V_1 > V_G \land V_2 \leq V_G) \land (w_1 \oplus w_2)$$

$$R_{p_0} = (V_1 > V_G \land V_2 > V_G) \land (w_1 \oplus w_2)$$

$$R_{p_1} = (V_1 + V_2 > V_G \land V_1 \leq V_G \land V_1 \leq V_2) \land (w_1 \oplus w_2)$$

$$R_{p_0} = (V_1 + V_2 > V_G \land V_1 \leq V_G \land V_2 > V_G) \land (w_1 \oplus w_2)$$

Similarly, $\neg (R \cup R_p)$ includes every state that is outside $S$ and more than one step is needed to reach a state in $S$. Therefore, state predicate $\neg (R \cup R_p)$ is the union of state predicates $R_{p_1}^\prime$ to $R_{p_0}^\prime$ as follows:

$$R_{p_1}^\prime = (V_1 + V_2 \leq V_G) \land (\neg w_1 \land \neg w_2)$$

$$R_{p_0}^\prime = (V_1 \leq V_G \land V_2 > V_G) \land (\neg w_1 \land w_2)$$

$$R_{p_1}^\prime = (V_1 > V_G \land V_2 \leq V_G) \land (w_1 \land w_2)$$

$$R_{p_0}^\prime = (V_1 > V_G \land V_2 > V_G) \land (w_1 \land w_2)$$

$$R_{p_1}^\prime = (V_1 + V_2 > V_G \land V_1 \leq V_G \land V_1 \leq V_2) \land (w_1 \land \neg w_2)$$

$$R_{p_0}^\prime = (V_1 + V_2 > V_G \land V_1 \leq V_G \land V_2 > V_G) \land (\neg w_1 \land w_2)$$

Now, observe that for any status of switches, there exists a state in $\neg (R \cup R_p)$. That means from any state in $S$, it is possible to reach a state in $\neg (R \cup R_p)$ without changing the value of switches using an environment transition. Therefore, no state is added to $R$ in the first iteration, and loop on Lines 4-14 terminates in the first iteration. Since, all the states outside $S$ remains in $\neg R$, the algorithm declares no solution to the addition problem exists. Therefore, according to the completeness of the Algorithm 1, there does not exist any $\delta_{b_2}$-safe stabilizing program for the smart grid described in this section when $k$ is equal to 2. This is expected since the only solution for this problem requires changing both sensors simultaneously before the environment is able to disrupt it again. This program does have a solution for $k=3$. But we omit its derivation for lack of space.

### 5. ADDITION OF FAULT-TOLERANCE

In this section, we present our algorithm for adding fail-safe fault-tolerance. In Section 5.1, we identify the problem statement for adding fault-tolerance, and in Section 5.2, we present the algorithm.

#### 5.1 Problem Definition

In addition to the set of bad transitions $\delta_b$ that we used for providing safe stabilization, in this case, we introduce additional parameter $\delta_e$ that identifies additional restrictions on program transitions. As an example, consider the case where a program cannot change the value of sensor, i.e., it can only read it. However, the environment can change the value of the sensor. In this case, transitions that change the value of both sensors are disallowed as program transitions but, they are acceptable as environment transitions. Note that this was not necessary in Section 3 since we could simply add these transitions to $\delta_e$, i.e., transitions that violate safety. This is acceptable since addition stabilization requires $\delta_b \cap \delta_e = \emptyset$. However, adding fault-tolerance is possible even if $\delta_b \cap \delta_e \neq \emptyset$. Hence, we add the parameter $\delta_e$ explicitly. The problem statement for addition of fault-tolerance is as follows:
proof, we assume that there are no deadlocks in $\delta_S$ state in $i.e.,$ states in $\delta_S$ms (as by any transition in $\delta_S$ms), of the addition problem defined in the Section 5.1, the set of computations of the revised program inside its invariant should be a subset of set of computations of the original program inside its invariant. Thus, the revised program cannot have any new computation starting from its invariant. In loop on Lines 13 - 22 we remove states from $S'$ to avoid creating such new computations.

Consider a state $s_0$ starting from which there exists environment transition $(s_0, s_1)$. In addition there exists program transition $(s_0, s_2)$ in the set of program transitions of the original program, $\delta_c$. 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 fairness assumption, $(s_0, s_1)$ cannot occur. Thus, sequence $(s_3, s_0, s_1)$ cannot be in any computation of $p Sang$. Therefore, we should remove any state like $s_3$ from the invariant. Set $ms_4$ includes any state
Regarding completeness, the intuition is that if it were im-
and (input $\delta$). Note that outside $S'$, any program tran-
sition which is not in $mt$ is allowed to exists in the final
program. In Line 15, the algorithm declare the no solution
to the addition problem exists, if $S'$ is empty. Otherwise, at
the end of the algorithm, it returns $(\delta', S')$ as the solution
to the addition problem.

**Theorem 4.** Algorithm 2 is sound and complete. And,
its complexity is polynomial (in the state space of the pro-
gram).

For reasons of space, we provide the proofs in [20].

6. **EXTENSIONS OF ALGORITHMS**

In this section, we consider problems related to those ad-
dressed in Sections 3 and 5. Our first variation focuses on
Definition 5. In this definition, we assumed that the envi-
ronment is fair. Specifically, at least $k-1$ actions execute
between any two environment actions. We consider vari-
ations where (1) this property is satisfied eventually. In
other words, for some initial computation, environment ac-
tions may prevent the program from executing. However,
eventually, fairness is provided to program actions, and (2)
program actions are given even reduced fairness. Specifi-
cally, 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 re-
vised 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
$S' \subset S$.

**Changes to add stabilization and fault-tolerance with eventually fair environment.** No changes are
required to Algorithm 1 even if environment is eventually
fair. This is due to the fact that this algorithm constructs
programs that provide recovery from any state, i.e., it will
provide recovery from the state reached after the point when
fairness is restored. For Algorithms 2, we should change the
input $f$ to include $\delta_c \cup \delta_f$. 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 to add 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 to add stabilization and fault-tolerance based on relation between $S'$ (invariant of the fault-
tolerant program) and $S$ (invariant of the fault-intol-
erant 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 al-
gorithm ensures $S' = S$. Hence, it trivially satisfies $S' \subseteq S$.
Regarding completeness, the intuition is that if it were im-
possible to recover to states in $S$ then it is impossible to
recover to states that are a subset of $S$. Regarding Al-
gorithms 2, 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.

7. **RELATED WORK**

This paper focuses on addition of fault-tolerance prop-
erties in the presence of unchangeable environment actions.
This problem is an instance of model repair where some exist-
ing model/program is repaired to add new properties such as
safety, liveness, fault-tolerance, etc. Model repair with respect to
CTL properties was first considered in [7], and abstraction techniques for the same are presented in [9].
In [14], authors focus on the theory of model repair for mem-
oryless LTL properties in a game-theoretic fashion; i.e., a re-
paired model is obtained by synthesizing a winning strategy
for a 2-player game. Previously [3], authors have considered
the problem of model repair for UNITY specifications [8].
These results identify complexity results for adding prop-
erties such as invariant properties, leads-to properties etc.
Repair of probabilistic algorithms has also been considered in
the literature [21].

The problem of adding fault-tolerance to an existing pro-
gram has been discussed in the absence of environment ac-
tions. This work includes work on controller synthesis [10,
13, 19]. A tool for automated addition of fault-tolerance to
distributed programs is presented in [5]. This work utilizes
BDD based techniques to enable 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 pro-
ces/components cannot be changed will result in unac-
ceptable solutions. At the same time, we anticipate that the
BDD-based techniques considered in this work will be es-
pecially valuable to improve the performance of algorithms
presented in this paper.

The work on game theory [15,17] has focused on the prob-
lem of repair with 2-player game where the actions of the sec-
ond player are not changed. However, this work does not ad-
dress the issue of fault-tolerance. Also, the role of the envi-
ronment in our work is more general than that in [15,17,18].
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 [6], authors have presented an algorithm for adding
repair to component based models. They consider the prob-
lem where we cannot add to the interface of a physi-
cal component. However, it does not consider the issue of
unchangeable actions of them considered in this work.

8. **APPLICATION FOR DISTRIBUTED AND
CYBER-PHYSICAL SYSTEMS**

We considered the problem of model repair for systems
with unchangeable environment actions. By instantiating
these environment actions according to the system under
consideration, this work can be used in several contexts. We
briefly outline how this work can be used in the context of
distributed systems and cyber-physical systems.

One instance of systems with unchangeable actions is dis-
tributed programs consisting of several processes. Consider
such a collaborative distributed program where some com-
ponents are developed in house and some are third party components. It is anticipated that we are not allowed to change third party programs during repair. In that case, we can model the actions of those processes as unchangeable environment actions, and use algorithms provided in this paper to add stabilization/fault-tolerance. Our work is directly useful in high atomicity contexts where processes can view the state of all components but can modify only their own. In low atomicity contexts where processes have private memory that cannot be read by others, we need to introduce new restrictions. Specifically, in this context, we need to consider the issue of grouping [5] where adding or removing a transition requires one to add or remove groups of transitions. In particular, if two states \( s_0 \) and \( s_0 \) differ only in terms of private variables of another process then including a transition from \( s_0 \) requires us to add a transition from \( s_0 \). Extending the algorithms in this context is beyond the scope of this paper.

Another instance in this context is a cyber-physical system. Intuitively, a CPS consists of computational components and physical components. One typical constraint in repairing these systems to satisfy new requirements is that physical components cannot be modified due to complexity, cost, or their reliance on natural laws about physics, chemistry etc. In other words, to repair a CPS model, we may not be allowed to add/remove actions which model physical aspects of the system. Therefore, using the approach proposed here, we can model such physical actions as unchangeable environment actions. After modeling the CPS, we can utilize the algorithms provided in this paper to add stabilization/fault-tolerance automatically, and be sure that the stabilizing/fault-tolerant models found by the algorithms do not require any change to physical components.

9. CONCLUSION

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 fail-safe fault-tolerance. These algorithms are sound and complete and run in polynomial time (in the state space). This was unexpected in part because environment actions can play both a collaborative and disruptive role.

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. Finally, as discussed in Section 8, these algorithms are especially useful for repairing CPSs as well as repairing distributed systems where only a subset of processes are repairable.

Acknowledgement

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

10. REFERENCES


