Automation of Fault-tolerant Graceful Degradation

Yiyan Lin*‡, Sandeep Kulkarni*‡, Arshad Jhumka†§
*Department of Computer Science and Engineering, Michigan State University, East Lansing, USA
‡{linyiyan,sandeep}@cse.msu.edu
†Department of Computer Science, University of Warwick, Coventry, UK
§arshad@dcs.warwick.ac.uk

***Abstract*—Traditionally, (nonmasking and masking) fault-tolerance has focused on ensuring that after the occurrence of faults, the program recovers to states from where it continues to satisfy its original specification. However, a problem with this limited notion is that, in some cases, it may be impossible to recover to states from where the entire original specification is satisfied. For this reason, one can consider a fault-tolerant graceful-degradation program that ensures that upon the occurrence of faults, the program recovers to states from where a (given) subset of its specification is satisfied. Typically, the subset of specification satisfied thus would be the critical/important requirements.**

In this paper, we initially focus on *automatically* revising a given fault-intolerant program into a fault-tolerant gracefully degrading program. Specifically, we propose a two-step approach: In the first step, we transform the fault-intolerant program into a graceful program. This program is guaranteed to satisfy only the given subset of specification (e.g., critical requirements). In particular, this step involves adding new behaviors that will satisfy the given subset of the specification. The second step involves utilizing the original program and the graceful program to obtain a fault-tolerant gracefully degrading program. We also develop an algorithm to transform the gracefully degrading program into a distributed gracefully degrading program. Afterwards, the second phase of our transformation can be applied to generate a distributed fault-tolerant gracefully degrading program. We showcase the algorithm with three different non-trivial case studies. Finally, we formalize the problem of multi-graceful degradation and propose an algorithm that solves it and we use a complex case study to showcase the viability of the approach. All the algorithms have polynomial time complexity in the size of the state space of the original program.

I. Introduction

We are increasingly dependent on highly available computer systems to provide fully functional and stable services. However, these programs are often subjected to new types of faults that are not considered in the original design. This may occur, for example, due to changing user requirements or due to deployment of the program in a new environment. It is especially important that the addition of such fault-tolerance be done correctly, i.e., the addition indeed provides the required resilience and preserves the original functionality.

Some of the existing approaches for automated addition of fault-tolerance include [1]–[3], where three different levels of fault-tolerance, namely nonmasking, masking and stabilizing, are considered. The common requirement for these levels of tolerance can be succinctly described with Figure 1(a), where S denotes the states where the program operates in the absence of faults and T denotes the states where the program may operate in the presence of faults. Fault-tolerance requires that if the program is perturbed to a state in T then it recovers to a state in S so that the subsequent behavior satisfies the program specification. (Masking fault-tolerance has an additional requirement that recovery from T to S must be safe whereas stabilizing fault-tolerance has an additional requirement that T must include all possible states.)

However, one limitation of these approaches is that in many scenarios, it is impossible for the program to recover to the original program behavior after faults occur, i.e., it may be impossible for the program to recover to states in S. In such scenarios, it is desirable that the program recovers to an *acceptable state* (cf. S' in Figure 1(b)) from where it satisfies some gracefully degraded behavior, i.e., behavior that is close to its original behavior.

The goal of this work is to develop algorithms where we begin with a program that satisfies its specification in the absence of faults (while being in states in S) and the desired relaxed specification in the presence of faults. The goal of the algorithm is to first identify a suitable S' and then construct a fault-tolerant program that (1) preserves the specification in the absence of faults and (2) recovers to states in S' after the occurrence of faults from where it satisfies the given relaxed specification.

To illustrate this goal further, consider a canonical example of a graceful behavior program where the original program provides core services and auxiliary services. However, after recovery, the program guarantees core services. In this context, the goal of the algorithm would be to begin with a program that satisfies the original specification in the absence of faults and desired requirements in the presence of faults (critical requirements) and construct a program that will (1) satisfy both critical and auxiliary requirements in the absence of faults and (2) provide recovery to states where it satisfies critical requirements.

The idea of such graceful degradation has been introduced in [4], where authors consider the specification to consist of a set of n properties. Subsequently, they consider desirable subsets (which are typically much less than 2^n subsets) of these n properties. To further illustrate the application of graceful degradation, we use the printer example considered in [4]. A printer system consists of computers sending printing tasks to a collection of printers. The tasks are organized in a queue and each printer executes a transaction in which it dequeues only one task and then prints it. In an ideal scenario (S in Figure 1) , one may prefer FIFO order for print requests, i.e., the task that is dequeued first is printed first. Thus, in an ideal setting, the specification requires that at most one

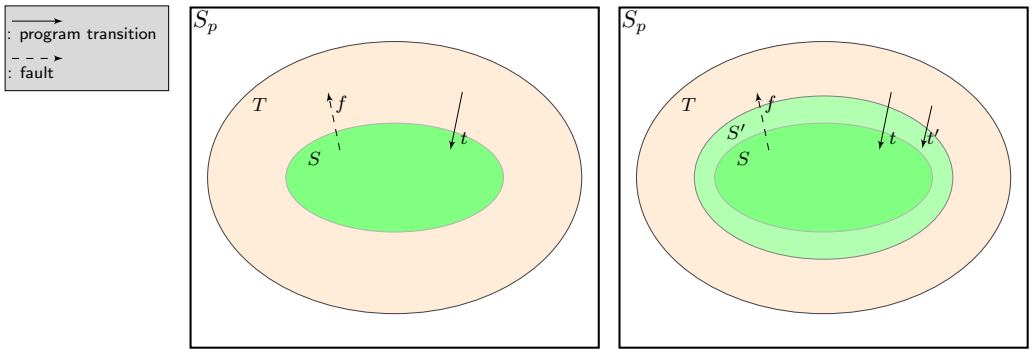

Fig. 1: Relationship between addition of graceful degradation and addition of fault-tolerant graceful degradation

dequeue operation can occur at a time. And, the next dequeue operation occurs only after the current task is printed.

For the sake of discussion, assume that in the presence of faults such as network delays or computer crashes, it may not be possible to guarantee that the program can recover to states from where this specification would be satisfied. Hence, one possible weaker specification (considered in [4]) is to allow a limited out-of-order printing. For example, one simple specification is that Task n is printed only after Task $n-2$ is printed although Task n may be printed before Task $n-1$.

In this example, the original program provides FIFO order for print requests. The desired fault-tolerant graceful degradation program has the following properties. In the absence of faults (states S in Figure 1), the program provides FIFO ordering. However, if faults occur then the program recovers to states (S' in Figure 1) from where it satisfies the degraded (weaker) specification (i.e., limited out-of-order printing).

One of the difficulties in generating the fault-tolerant program that provides graceful degradation lies in the fact that the input does not contain a program that satisfies the weaker specification (i.e., a program that satisfies out-of-order printing). Asking the designer to specify such a program is undesirable, since it increases the overhead for the designer. With this motivation, our approach for adding fault-tolerant graceful-degradation consists of two steps. The first step (Figure 2) focuses on the automated addition of graceful degradation to the fault-intolerant program in the *absence* of faults.

In particular, this step begins with a program, say p, that satisfies the original specification (FIFO order for the printer example) and constructs a program, say p_g, that satisfies the weaker, degraded specification (limited out-of-order printing). The program p_g is obtained by adding new behaviors to p, outside of S but within S' (e.g., to provide out-of-order printing in the printer example). Thus, in the case of the printer example, the generated program will provide FIFO behavior from the original states as well as limited out-of-order printing from additional states.

In the second step (Figure 3), we begin with program p and p_g to construct a fault-tolerant gracefully degrading program p_f. Program p_f ensures that in the absence of faults, it behaves like p and, hence, it will satisfy the original specification in the absence of faults. Moreover, if faults occur then p_f recovers to states from where p_g satisfies the degraded specification. Depending upon what p_f does during the recovery process, we can get different variations. For example, one variation can be similar to masking fault-tolerance where some safety properties are satisfied during the recovery process. Another variation can be similar to stabilizing fault-tolerance where recovery must be provided from an arbitrary state. [1]

Subsequently, we extend the problem of fault-tolerant graceful degradation to address the problem of fault-tolerant multi-graceful degradation. Intuitively, fault-tolerant multi-graceful degradation allows a hierarchy of severe faults and require that the program satisfy corresponding hierarchy of weaker specifications. After formalizing the problem of addition of fault-tolerant multi-graceful degradation to a program, we provide an algorithm, based on the two-step approach, that solves the problem. We illustrate the working of the algorithm through a detailed case study associated with Ohio Coal Research Center (OCRC).

Contributions of the paper. The contributions of the paper are as follows:

- We define the problem of automated addition of graceful degradation and present a polynomial-time algorithm in the size of the state space of the fault-intolerant program to solve it. Unlike previous algorithms [5] that focus on removing behaviors in the absence of faults, this algorithm focuses on adding new behaviors while ensuring that these behaviors still satisfy the weaker specification.
- We provide an algorithm that transforms a graceful program into a distributed graceful program.
- We adapt existing algorithms for adding fault-tolerance to design fault-tolerant graceful degradation.
- We illustrate our algorithm with three case studies, namely (i) the printer system, (ii) resource constraint problem motivated by channel assignment in cellular networks and (iii) the classic byzantine agreement problem.
- We extend the problem of adding graceful fault-tolerance

[1] Note that these variations are not the same as masking and/or stabilizing tolerance since p_f may not satisfy its original specification after recovery is complete. Moreover, if we utilize existing algorithms [5] where the input consists of program p then those algorithms will declare failure to add masking fault-tolerance if it is impossible to guarantee recovery to behaviors of program p.

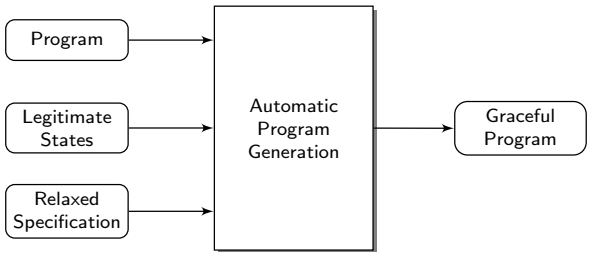

Fig. 2: Step 1: Design of Graceful Program

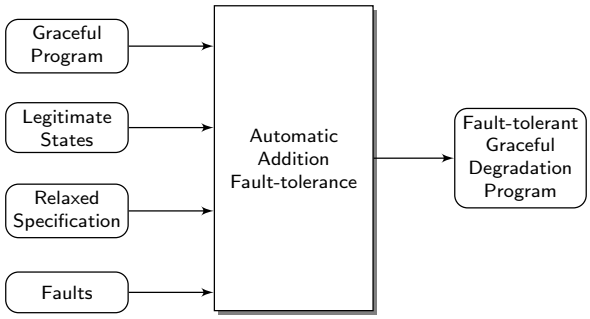

Fig. 3: Step 2: Design of Fault-Tolerant Graceful Degradation

to the case where we have a hierarchy of increasingly severe faults and we need to satisfy decreasing level of specification in their presence. For this purpose, we define the problem of automated addition of fault-tolerant multi-graceful degradation and present a polynomial-time algorithm, based on the two-step approach, with complexity in the size of the state space of the fault-intolerant program to solve it.

– We illustrate the multi-graceful degradation addition algorithm through a detailed case study of the Ohio Coal Research Center (OCRC) [6].

Organization of Paper. The rest of paper is organized as follows: We define the notion of program, specification and specification relaxation in Section II. In Section III, we formally state the problem of automated generation of the graceful program. In Section IV, we present the algorithm to generate the graceful program from the original program and the specification (original and relaxed). In Section V, we present two case studies to demonstrate the graceful program generation step by step. In Section VI, we discuss how to extend the algorithm for generating graceful programs to distributed programs. In Section VII, we use byzantine agreement as an example to further illustrate the technique of generating graceful distributed programs. In Section VIII, we define the problem of adding fault-tolerant graceful degradation and continue with the three case studies to add fault-tolerance to the programs generated in Sections V and VII. We then present and formalize the problem of automated addition of multi-graceful degradation in Section IX and present a polynomial-time algorithm that solves it. To show the working of the multi-graceful degradation addition algorithm, we present the OCRC system in detail in Section X, present a model of the system and then show the application of the multi-graceful degradation algorithm to the OCRC system. Finally, we discuss related work in Section XI and provide further insights and a summarize the paper in Section XII, where we also provide insights of our ongoing work.

II. Preliminary

In this section, we give formal definitions of programs, program specifications, and graceful degradation. The program is specified in terms of its state space and transitions. The definition of specification is adapted from [7]. The notion of graceful degradation is adapted from [4].

A. Program

A program p, specified as a tuple $\langle S_p, \delta_p \rangle$, consists of its finite state space S_p and transitions δ_p, where $\delta_p \subseteq S_p \times S_p$. A state predicate of p is any subset of S_p. A state predicate S is closed in p (respectively, δ_p) iff $(\forall (s_0, s_1) \in \delta_p : (s_0 \in S \Rightarrow s_1 \in S))$. A sequence of states, $\langle s_0, s_1, \cdots \rangle$ (denoted by σ), is a computation of p iff(1) $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \in \delta_p$, and (2) if σ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in \delta_p$. In other words, in each step of the computation of p, some transition of p is executed. And, the computation is finite iff p does not have any transition in the final state.

The projection of program p on state predicate S, denoted as $p|S$ is the program $\langle S_p, \{(s_0, s_1) \in \delta_p \wedge s_0, s_1 \in S\} \rangle$. In other words, $p|S$ includes transitions of p that begin and end in S.

Notation. If the context is clear, we use p and δ_p (transitions of p) interchangeably. Also, we say that a state predicate S is true in state s iff $s \in S$.

B. Program Syntax

To model the state space and transitions concisely, we use processes, variables (associated with a finite domain) and actions. The state space S_p of p is obtained by assigning each variable a value from its respective domain. Thus, the use of variables allows us to represent the state space compactly. Additionally, to compactly represent δ_p, we use actions of the form: $guard \longrightarrow statement$ where $guard$ is a constraint (predicate) involving program variables and $statement$ updates program variables [8]. We denote variable x of a process j by $x.j$. An action $guard \longrightarrow statement$ denotes the set of transitions $\{(s_0, s_1) : guard$ is true in s_0 and s_1 is obtained by changing s_0 as prescribed by $statement\}$. We use $x(s)$ to denote the value of variable x in state s. This compact representation has been used in several past works to represent distributed algorithms, e.g., [8].

As an example, consider a triple modular redundancy (TMR) system, with three inputs x, y and z to the voter and an output out from the voter. One action of the TMR system is as follows:

$$out = \bot \wedge x = y = z \rightarrow out := x$$

This action basically states that when all three inputs are the same, the voter chooses one of them as output. To further

illustrate that an action provides a concise representation of transitions, consider the domain of x, y and z to be $\{0, 1\}$ and out is defined over $\{0, 1, \perp\}$, where $\perp$ denotes an undefined value. The state space of the TMR program will contain 24 states. The above action will represent the following set of transitions:

$$\begin{aligned}\{(s,t)| \quad & ((x(s) = y(s) = z(s) = 0 \wedge out(s) = \perp) \wedge \\ & (x(t) = y(t) = z(t) = 0 \wedge out(t) = 0)) \\ \vee \quad & \\ & ((x(s) = y(s) = z(s) = 1 \wedge out(s) = \perp) \wedge \\ & (x(t) = y(t) = z(t) = 1 \wedge out(t) = 1)) \\ \}.\end{aligned}$$

C. Specification

Following Alpern and Schneider [7], we let the specification of program consist of a safety specification and a liveness specification. The safety specification is specified in terms of a set of bad states, say $spec_{bs}$, that program is not allowed to reach, and a set of bad transitions, $spec_{bt}$, that the program is not allowed to execute. Thus, a sequence $\langle s_0, s_1, \cdots\rangle$ (denoted by σ) satisfies the safety specification iff (1) $\forall j : 0 \leq j < length(\sigma) : s_j \notin spec_{bs}$, and (2) $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \notin spec_{bt}$.

The liveness specification, on the other hand, denotes "good thing" happens during program execution. We use $leadsto$ property ($\mathcal{L} \rightsquigarrow \mathcal{T}$) to denote liveness specification, where both $\mathcal{L}$ and $\mathcal{T}$ are state predicates. Thus, a sequence $\langle s_0, s_1, \cdots\rangle$ (denoted by σ) satisfies the liveness specification iff $\forall j : (\mathcal{L}$ is true in $s_j \Rightarrow \exists k : j \leq k < length(\sigma) : \mathcal{T}$ is true in $s_k)$.

A *specification*, say $spec$, is a tuple $\langle Sf_p\ ,\ Lv_p\rangle$, where Sf_p is a safety specification and Lv_p is a liveness specification. A sequence σ *satisfies* $spec$ iff it satisfies Sf_p and Lv_p. Hence, for brevity, we say that the program specification is an intersection of a safety specification and a liveness specification.

Given a program p, a state predicate S, and specification $spec$, we say S is an invariant of p iff (1) S is closed in p; (2) Every computation of p that starts in a state, say s, where $s \in S$ satisfies $spec$; and (3) $S \neq \emptyset$.

D. Graceful Degradation

In graceful degradation, it is expected that the program will satisfy a stronger specification under normal circumstances. And, it will satisfy a weaker specification under some other circumstances (e.g., in the presence of faults). Let $spec = \langle Sf, Lv\rangle$ and $spec_r = \langle Sf_r, Lv_r\rangle$ be two specifications. We say that $spec_r$ is weaker than $spec$ iff for any sequence σ if σ satisfies $spec$ then σ satisfies $spec_r$.

III. Problem Statement for Graceful Degradation Design

In this section, we formally define the problem of generating a program that satisfies the weaker specification (cf. Figure 2). We begin with a program p that satisfies the specification $spec$ from invariant I. We derive a graceful degrading program, say p_g, and its invariant I_g such that p_g satisfies a weaker specification, say $spec_r$ from I_g. Next, we consider the relation between p and p_g as well as I and I_g to identify the problem statement. One requirement on p_g is that p_g is supposed to add new behaviors that potentially violate $spec$ while satisfying $spec_r$. And, it is not allowed to remove any behaviors of p that satisfy $spec$. Since the correctness of p is known from its invariant I, based on this requirement, it follows that I should be a subset of I_g. Additionally, p_g cannot remove any behaviors (respectively, transitions) within the original invariant I. Thus, the problem statement is shown as follows:

> **Problem Statement III.1:**
> Given p, I, $spec$ and $spec_r$ such that p satisfies $spec$ from I. Identify p_g and I_g such that:
> **A1**: $p_g|I = p|I$, $I \subseteq I_g$.
> **A2**: p_g satisfies $spec_r$ from I_g.
> **A3**: $p_g|I$ satisfies $spec$ from I.

Remark. The above problem statement has a requirement that I be a subset of I_g. This requirement differs from [5] in that this enlarges the invariant by adding new states from where the new specification can be satisfied. By contrast, in [5], the problem statement requires that the generated invariant be a subset of the original invariant. For this reason, existing algorithms cannot be used to solve the above problem.

IV. Algorithm for Generating Graceful Program

We begin with the given program p, its invariant I and its two specifications (stronger) $spec$ and (weaker) $spec_r$. In turn, $spec$ and $spec_r$ are specified in terms of the corresponding safety and liveness specification. Our goal is to construct p_g and I_g such that they satisfy constraints of Problem III.1.

Algorithm 1 also takes an additional input which is a state predicate, namely S_a. The intuition for S_a arises from the fact that the program is expected to satisfy the specification $spec_r$ under certain constraints. Predicate S_a is used to characterize these constraints. If such constraints are not easily identifiable, S_a can be instantiated to be $S_p - I$, i.e., all states except those in I.

First, Algorithm 1 computes I_Δ to be the set of states in S_a except those that violate safety (i.e., those in $spec_{r_{bs}}$). The first guess for I_g, the invariant for the graceful program, is then set to $I \cup I_\Delta$ (Line 2). Then, Algorithm 1 computes the first guess for p_g. Specifically, in p_g, we reuse all the transitions in p that begin in I. We also include all transitions that begin in I_Δ unless they violate safety $spec_{r_{bt}}$ (Line 3). Starting from Line 4, the algorithm revises the program by ensuring the liveness specification is satisfied. In Lines 6-9, we exclude the deadlock states, i.e., states from where no transitions originates from I_g. Then we recompute p_g such that all the reachable states by p_g starting from I'(or I) remains in I'(or I respectively). To ensure liveness, we define a function $rank$ that assigns each state an integer value that represents the length of the shortest path from that state to reach a target state predicate $\mathcal{T}$. Then, on Lines 11 and 12, we exclude states and transitions where rank does not decrease. Removal of such transitions ensures that there will be no cycles that prevent the program from reaching $\mathcal{T}$. To resolve deadlock states, we repeat the loop

starting at Line 4. Finally, this process stops once a fixpoint is reached.

Algorithm 1 Graceful Program Generation

Input: state predicate S_a, program transitions p, invariant I, weaker safety specification Sf_r (consisting of $Sf_{r_{bs}}$ and $Sf_{r_{bt}}$), liveness specification Lv(consisting n leads-to properties of the form $\mathcal{F}_i \rightsquigarrow \mathcal{T}_i$, $i \in 1 \cdots n$).

Output: graceful program p_g and invariant I_g with weaker specification.

1: $I_\Delta := S_a - Sf_{r_{bs}}$
2: $I' := I \cup I_\Delta$
3: $p' := \{(s_0, s_1) | s_0, s_1 \in I' :: (s_0 \in I \wedge (s_0, s_1) \in p) \vee (s_0 \in I_\Delta \wedge (s_0, s_1) \notin SPEC_{r_{bt}})\}$
4: **repeat**
5: $\quad I_{old} := I', p_{old} := p'$
6: $\quad$ **repeat**
7: $\quad\quad I_{old} := I'$
8: $\quad\quad p' := maxp(p', I, I')$
9: $\quad\quad I' := I' - deadlock(I', p')$
10: $\quad$ **until** $I_{old} = I'$
11: $\quad p' := p' - \bigcup_{i \in 1 \cdots n} \{(s_0, s_1) | s_0 \in I' \wedge rank(s_0, \mathcal{T}_i, p') > rank(s_1, \mathcal{T}_i, p') \wedge rank(s_0, \mathcal{T}_i, p') \neq 0 \wedge rank(s_1, \mathcal{T}_i, p') \neq \infty\}$
12: $\quad I' := I' - \bigcup_{i \in 1 \cdots n} \{s | rank(s, \mathcal{T}_i, p') = \infty \wedge s \in \mathcal{F}_i \wedge s \notin I\}$
13: **until** $I_{old} = I' \wedge p_{old} = p'$
14: **return** p' as p_g, I' as I_g if $I' \neq \emptyset$, otherwise declare no graceful program generated.

Function: $maxp$(t: transition predicate, S_1,S_2,$\cdots$:set of state predicates where $S_1 \subseteq S_2 \subseteq \cdots$)
return $\{(s_0, s_1) | t \cap (s_0 \in S_1 \Rightarrow s_1 \in S_1) \wedge (s_0 \in S_2 \Rightarrow s_1 \in S_2) \wedge \cdots\}$

Function: $deadlock$(S: state predicate , t: transition predicate)
return $\{s_0 | s_0 \in S \wedge (\forall s_1 \in S : (s_0, s_1) \notin t)\}$

Function: $rank$(s: state , $\mathcal{T}$: state predicate , t: transition predicate)
return the shortest path length from s to one of the state in $\mathcal{T}$, if the path(consisting only transitions in t) exists; or ∞, otherwise.

We now prove the correctness of Algorithm 1.

Lemma 1 (Correctness of Algorithm 1)**.** *Let the input to Algorithm 1 be:*

- *program p*
- *invariant I*
- *spec (original specification)*
- *spec$_r$ (relaxed specification)*
- *S_a (state predicate)*

Let the output of Algorithm 1 be:

- *graceful program p_g*
- *weaker invariant I_g*

Then

1) $I \subseteq I'$
2) $p' \mid I = p \mid I$
3) $p' \mid I$ *satisfies* spec *from* I
4) p' *satisfies spec$_r$ from* I'

Proof:

We proceed by proving each case.

1) $I \subseteq I'$: From Line 2, there are two cases to consider: (a) $I' = I$ and (b) $I \subset I'$.
   a) $I' = I$: On Line 9, I' remains the same since there is no deadlock state in the invariant of the program. On Line 12, I' does not change. Hence, $I' = I$ through out.
   b) $I \subset I'$: On Line 9, $I \subseteq I'$ since deadlock states $\subseteq (I' - I)$. On Line 12, only states $s \in I' - I$ are removed. Hence, $I \subseteq I'$.

   From 1a and 1b, $I \subseteq I'$.
2) $p' \mid I = p \mid I$: From Line 3, $(s_0 \in I \wedge (s_0, s_1) \in p)$ implies that $p' \mid I = p \mid I$. On Line 8, computation of *maxp* does not result in any state or transition in the invariant being removed. Hence, the predicate $p' \mid I = p \mid I$ holds. On Line 11, the set of transitions that are removed are not in I. Hence, the condition holds through out.
3) $p' \mid I$ satisfies *spec* from I: Since p satisfies *spec* from I and none of the transitions that begin in I are removed, then $p'|I = p|I$. Hence, p' satisfies *spec* from I.
4) p' satisfies *spec$_r$* from I' : Closure of I' follows from Line 8, where transitions that can potentially violate the closure are removed in $maxp$. Also, by construction, p' cannot reach a state in $spec_{r_{bs}}$ and cannot execute a transition in $spec_{r_{bt}}$ (Lines 3, 8 and 11). Finally, by Lines 11 and 12, the liveness property is also satisfied. Hence, p' satisfies $spec_r$ from I'.

■

It can be noted that Algorithm 1 has a state predicate S_a as input. The above lemma is valid for any input value of S_a as it is only used to constrain the search space, if the designer is already aware of constraints that would be met even during graceful behavior. If the designer is not aware of such constraints, S_a cannot be constrained and the whole state space needs to be searched, i.e., S_a can be set to be equal to $S_p - I$. Next, we prove that Algorithm 1 terminates.

Lemma 2 (Termination of Algorithm 1)**.** *Let the input to Algorithm 1 be:*

- *program p*
- *invariant I*
- *spec (original specification)*
- *spec$_r$ (relaxed specification)*
- *S_a (state predicate)*

Then, Algorithm 1 terminates in polynomial time in the state space of p.

Proof:

We need to show that the two loops (Lines 6-10 and 4-13) terminate. We prove this by showing the existence of a least fixpoint for both loops.

- Inner loop (Lines 6-10): The loop terminates when $I_{old} = I'$. As I' becomes smaller with each iteration (Lines 8 and 9) and since $I' \subseteq I_{old}$, the loop will definitely terminate when $I' = \emptyset$.
- Outer loop (Lines 4-13): The loop terminates when $(I_{old} = I') \wedge (p_{old} = p')$. As I' and p' become smaller with each iteration (Lines 11 and 12) and since $(I' \subseteq I_{old}) \wedge (p' \subseteq p_{old})$, the loop will terminate.

■

Theorem 1. *Let the input to Algorithm 1 be: (i) program p, (ii) invariant I, (iii) original specification spec, (iv) $spec_r$ (relaxed specification) and (v) S_a (state predicate)*

Then, Algorithm 1 solves problem III.1 *in polynomial time.*

Proof: It follows directly from Lemmas 1 and 2. ■

V. Graceful Degradation Design: Case Studies

In this section, we demonstrate the process of generating the gracefully degrading programs through two case studies, namely (i) a printer system and (ii) a resource constraint problem motivated by cellular networks. In each case study, we first define the program with its specification (including the original and the graceful specification). Then, we simulate the algorithm on each case study. Subsequently in Section VIII, we revisit them in the context of adding fault-tolerance.

A. Printer System

In the first case study, we focus on the printer system considered in [4]. In the original specification, the printer system is required to satisfy the $FIFO$ specification. However, as argued in [4], it may be necessary to relax this requirement under certain constraints. In particular, the weaker requirement considered in [4] requires that some out-of-order printing is permitted.

Original program. The program in [4] consists of several clients and print servers. Using the transaction semantics, the clients 'enqueue' their print requests in a central queue. A print server then removes a task from this non-empty queue and prints the task. Please observe that the queue is of bounded length as we assume finite state programs. For simplicity, we do not model the transactions used for concurrency control and the enqueue operation since it does not affect the behavior we are interested in. The original program, say $pr_{intolerant}$, consists of two parts (1) once i^{th} task has been dequeued ($d_i = 1$) and it has not been printed ($p_i = 0$), then print it ($p_i := 1$) and (2) if the i^{th} task has been printed ($p_i = 1$), then the next Task $(i + 1)$ in queue gets a chance to be dequeued ($d_{i+1} := 1$). Thus actions are as follows:

$$\begin{aligned} d_i = 1 \wedge p_i = 0 &\longrightarrow \quad p_i := 1 \\ p_i = 1 &\longrightarrow \quad d_{i+1} := 1 \end{aligned}$$

For sake of simplicity, in this paper, we consider the case where the number of jobs is enumerated explicitly. To make this more generic, one can utilize approaches for parametric synthesis where the number of jobs is left as a parameter. Examples of such parametric approaches are discussed in [9], [10].

Safety Specification. Recall that the safety specification is in terms of a set of bad states that the program should not reach and a set of bad transitions that the program should not execute. The first specification Sf_{bs_1} is satisfied by the original program and requires that the printing must occur in order and, hence, Task j cannot be dequeued until Task $j-1$ is printed. The graceful specification permits the possibility that Task j can be dequeued even if Task $j-1$ is not printed. However, it requires that Task j cannot be dequeued until Task $j-2$ is printed. Thus, the safety specification (original and graceful) that identifies the states that should not be reached is specified as follows:

$Sf_{bs\_1} := \exists i, j \in 1 \cdots n, p_i(s) = 0 \wedge d_j(s) = 1 \wedge j > i$

$Sf_{bs\_2} := \exists i, j \in 1 \cdots n, p_i(s) = 0 \wedge d_j(s) = 1 \wedge j > i+1$

Observe that Sf_{bs_1} is the stronger specification capturing the notion of FIFO, and Sf_{bs_2} is a weaker specification since it allows at most two tasks be dequeued without finishing the pending printing. One could consider further relaxation that allows more out-of-order printing. Our algorithm can be applied in this context by applying Algorithm 1 to the program described at the end of this section. However, the detailed analysis of this generation is outside the scope of this paper.

In addition to bad states, safety specification can include bad transitions that the program is not allowed to execute. The bad transitions, Sf_{bt_c}, describe *structural constraints* that have to be satisfied by the printer system. In particular, Sf_{bt_c} states that a printed task cannot be reset to unprinted. Likewise, once the task has been dequeued it cannot be re-enqueued. And, a task cannot be printed until it is dequeued. Note that the structural constraints have to be satisfied in the original as well as the graceful program. Also, at most one variable can be changed in any transition. Thus,

$$\begin{aligned} Sf_{bt\_c} := \{(s, s') | \exists i, j \in 1 \cdots n, \\ & (p_i(s) = 1 \wedge p_i(s') = 0) \\ & \vee (d_i(s) = 1 \wedge d_i(s') = 0) \\ & \vee (d_i(s') = 0 \wedge p_i(s') = 1) \\ & \vee (i \neq j \wedge d_i(s) \neq d_i(s') \wedge d_j(s) \neq d_j(s')) \\ & \vee (i \neq j \wedge p_i(s) \neq p_i(s') \wedge p_j(s) \neq p_j(s')) \\ & \vee (d_i(s) \neq d_i(s') \wedge p_j(s) \neq p_j(s'))\} \end{aligned}$$

We use safety specification for original program as Sf_{bs_1} and Sf_{bt_c}; for graceful program as Sf_{bs_2} and Sf_{bt_c}. Note that although the set of bad transitions are the same here, it is not a requirement for Algorithm 1.

Liveness Specification. The liveness specification requires that eventually all tasks are printed. Thus,

$Lv := true \rightsquigarrow S_L$, where $S_L := \forall i \in 1 \cdots n, d_i = p_i(s) = 1$.

Application of Algorithm 1. We instantiate Algorithm 1 with following inputs:

- program p: is instantiated to be transitions corresponding to program $pr_{intolerant}$. For simplicity, we assume that there are three tasks in the system. Hence, the state of the program is represented as $(d_1d_2d_3, p_1p_2p_3)$, where d_i (respectively, p_i) denotes whether Task i has been dequeued (respectively printed).
- The safety specification is instantiated so that the set of bad states is Sf_{bs_2} and the set of bad transitions is Sf_{bt_c} .
- Liveness specification is specified to be Lv.
- Invariant I is instantiated to be the states reached in the computation of $pr_{intolerant}$ by starting from the initial state $(000, 000)$.
- State predicate S_a is instantiated to be $S_p - I$

Consider the candidate states generated outside I from Line 1 that also exclude those violate Sf_{bs_2} . Recall that Sf_{bs_2} allows two tasks to be dequeued at once and allows one task to be printed before the previous task is completed. Thus, we have

$I_\Delta = \{(110,000),(110,010),(111,010),(111,011),$
$(110,110),(111,100),(111,101),(010,000),(010,010),$
$(011,010),(011,011)\}$

The new program transitions generated are all possible pairs of states in I_Δ plus those that provide recovery from I_Δ to I provided that they do not violate Sf_{bt_c}. There are no deadlock states on Line 8. Moreover, since every state in I or I_Δ has a path to a state where all tasks are completed, no state and transition are removed on Lines 11 and 12.

Now, we evaluate the new behavior of the printer system in p_g provided in Figure 4. Ideally, the program stays in invariant I while satisfying Sf_{bs_1}, Sf_{bt_c} and Lv. However, if the program is perturbed to be outside I then it will satisfy Sf_{bs_2}, Sf_{bt_c} and Lv. Figure 4 shows the behavior of the graceful program. Note that in this program, the tasks being printed are not totally out of order, though it is not FIFO. For example, Task 3 can be printed before Task 2 but Task 3 cannot be printed before Task 1.

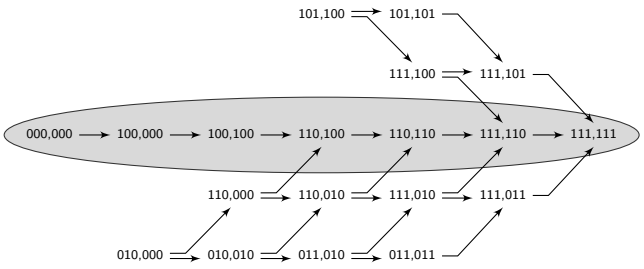

Fig. 4: Printer System with original invariant $I = \{(000,000),(100,000),(100,100),(110,100),(110,110),(111,110),(111,111)\}$ and invariant (under weaker specification) $I_g = I \cup \{(110,000),(110,010),(111,010),(111,011),(110,110),(111,100),(111,101),(010,000),(010,010),(011,010),(011,011)\}$

B. Resource Constraint Problem: Channel Assignment

In this section, we model the problem of resource constraints in a wireless cellular network [11]. When a call is made, a number of channels are assigned to it. However, due to the *finite* number of channels available, these have to be shared among a finite number of calls, and call degradation can be incorporated to maximize the revenue by maximizing the number of admitted calls. For brevity, we consider a simplified version of this problem and describe how Algorithm 1 can be applied to generate a graceful program.

In such a network, each cell is associated with a set of frequencies. Each frequency can be used by a single call. However, to improve quality of the call, one call may be assigned several frequencies. Clearly, if a call is assigned 0 channels/frequencies then the call cannot be completed and the call fails. As the number of assigned channels increases the call quality improves. There is a minimum number of channels (MIN) that must be allocated to a call so that the call quality is at an acceptable level. Also, there is a maximum number of channels (MAX) that should be allocated, as allocating subsequent channels does not lead to improvement in call quality.

From a user perspective, to keep user satisfaction at higher levels, it is desirable to assign as many channels as possible to every call, up to MAX channels. And, from a system perspective, it is desirable to assign as few channels as possible so that the probability of a call failure (where the call is dropped due to unavailable channels) is kept very low. To simplify the modeling of this system, we consider the case where the maximum channels allocated to a call is 3 ($MAX = 3$) and the minimum number of channels allocated is 2 ($MIN = 2$).

Normally, each call in such a system is defined by the following parameters:

- Channel requirements: Each call defines its preferred (or maximum) number of channels.
- Priority: This determines the class/type of the call.
- Degradation tolerance: The (maximum) number of channels that can be reclaimed from the call.
- Admission policy: If a cell becomes saturated, it can either reject an incoming call or degrade ongoing calls to accommodate new calls.
- Revenue: Each admitted call generates a revenue based on the agreed QoS (i.e., number of channels).

For the purpose of keeping the case study simple, we consider all calls to be of the same priority and all calls have a degradation tolerance of 1 ($MAX - MIN$). For the admission policy, we consider call degradation (for gracefully degrading calls). We do not model revenue as such, but try to maximize the number of admitted calls. Furthermore, we also assume that there is a finite number of channels and calls and that calls are of infinite duration, i.e., reclamation of channels is omitted for simplicity. We note that these assumptions are only for simplicity and Algorithm 1 can be easily applied in cases where these assumptions are omitted.

Original program The system begins with a given number ($MAXC$) of available channels. A call is admitted to the system only if it can be allocated MAX (3) channels. Otherwise, the call fails. To model such a program, for each call, we maintain a variable c_i that denotes whether call i is current (i.e., admitted) and variable a_i that denotes the number of channels allocated to call i. Instead, we assume that actions for allocating channels to call i are executed only when call i is requesting them. We also use variable av to denote the number of available channels. This variable is initialized to $MAXC$ in the initial state.

$$c_i = 0 \wedge av \geq 3 \longrightarrow \quad c_i, a_i, av = 1, 3, av - 3$$

Original safety property

The original safety specification requires that each admitted call is either allocated three channels or no channel. It also requires that the total number of allocated and unallocated channels equal to $MAXC$. Thus, the safety specification of the original program is:

$$\begin{aligned} Sf_{bs\_1} := \quad & (\exists i : c_i = 0 \wedge a_i \neq 0) \\ \vee \quad & (\exists i : c_i = 1 \wedge a_i \neq 3) \\ \vee \quad & (\textstyle\sum_{i=1}^{n} a_i + av \neq MAXC) \end{aligned}$$

where n is the number of admitted calls.

Also, since we are not modeling release of channels, we say that the following transitions – where we start in a state where a call has been assigned some channels and end up in a state where the call has been assigned less channels – also violate safety.

$$Sf_{bt\_c} := (\exists i :: c_i = 1 \wedge c_i' = 0) \vee (av' > av)$$

Original liveness Specification.

The liveness specification requires that all calls are serviced when channels are available. Thus, the liveness specification is as follows

$$Lv := true \rightsquigarrow ((\forall i :: c_i = 1) \vee av < 3)$$

Relaxed safety property.

Observe that the original program requires that each active call is assigned three channels. Based on the earlier discussion, the relaxed program is permitted to assign two channels to a call or 1 channel can be reclaimed from ongoing calls. However, since the goal of the channel assignment program is to assign two channels only if required, we capture this with the following relaxed safety property. Intuitively, this relaxed safety property requires that if some call is assigned two channels, it is due to the fact that it could not be assigned the three possible channels. Hence, the states that should not be reached in the graceful program are

$$\begin{aligned} Sf_{bs\_2} := (\exists i :: c_i = 1 \wedge ((a_i \neq 3) \vee (a_i = 2 \wedge av > 0))) \\ \vee (\textstyle\sum_{i=1}^{n} a_i + av \neq MAXC) \end{aligned}$$

Application of Algorithm 1.

We instantiate Algorithm 1 with the following inputs:

- The set of program transitions is instantiated with those associated with the original program, denoted by P_c in subsequent description. For illustration, we assume that the total available channels is $MAXC = 7$. We represent the state of P_c as $(av, c_1c_2c_3 \cdots c_n, a_1a_2a_3 \cdots a_n)$, where c_i denotes whether call i is admitted, a_i denotes the number of channels allocated to call i and av denotes the number of remaining channels. For illustration purpose, we consider the case where the number of calls is 3, i.e., $n = 3$.
- The invariant I is instantiated to be states reached from the initial state where all (7) channels are available and no channel is allocated to any call. Thus, I is the set of states reached from the state $(7, 000, 000)$ in the original program.
- The state predicate S_a, is instantiated to be $S_{P_c} - I$ (set of states of P_c except the states in the invariant).
- We instantiate the safety specification with Sf_{bs_2}, the set of bad states for the graceful program, and Sf_{bt_c}, the set of bad transitions.
- The liveness specification is instantiated with Lv.

Since the relaxed safety specification requires that a call is assigned two channels only if no free channels are available, the only states in S_a from where two channels are allocated to a process are those where the number of free channels is either 0, 1 or 2. Moreover, since the sum of the assigned and free channels is equal to $MAXC$, any action added to the relaxed program can only redistribute channels among processes. Hence, the relaxed program includes the following three actions that correspond to the case where the number of available channels is either 0, 1 or 2. It also includes actions where we only switch between channels assigned to a single call. Thus, the actions are as follows:

$$\begin{aligned} & c_i = 0 \wedge a_i = 0 \wedge av = 0 \wedge c_j = c_k = 1 \wedge a_j = a_k = 3 \\ \longrightarrow\ & c_i, a_i, a_j, a_k := 1, 2, 2, 2 \end{aligned}$$

$$\begin{aligned} & c_i = 0 \wedge a_i = 0 \wedge av = 1 \wedge c_j = 1 \wedge a_j = 3 \\ \longrightarrow\ & c_i, a_i, a_j, av := 1, 2, 2, 0 \end{aligned}$$

$$\begin{aligned} & a_i = 0 \wedge av = 2 \\ \longrightarrow\ & c_i, a_i := 1, 2 \end{aligned}$$

$$\begin{aligned} & c_i = c_j = 1 \wedge a_i = 2 \wedge a_j = 3 \\ \longrightarrow\ & a_i, a_j := 3, 2 \end{aligned}$$

where i j and k are quantified over the number of calls in the system.

We can also visualize the above actions as shown in Figure 5.

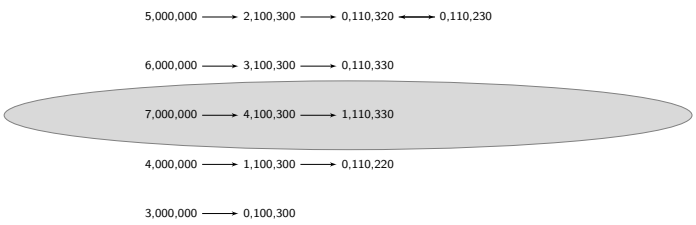

Fig. 5: Cellular Networks: MAXC = 7, MAX = 3, MIN = 2, Degradation tolerance = 1.

VI. Algorithm for Generating Distributed Graceful Programs

In this section, we extend Algorithm 1 (see Section IV) to handle distributed programs. In particular, Algorithm 1 focuses on deriving a concurrent graceful program, where the program is able to read and update all variables in an atomic step. In applying Algorithm 1 to distributed programs, it is necessary that the generated program can be implemented using a set of processes.

The main difference between a concurrent program and a distributed program is that, in a distributed program, each process may have some *private* variables to which they have sole access. Use of such variables is essential in ensuring that the graceful program is implementable in a distributed system. As an illustration, if we model processes that may be byzantine, then the knowledge of whether a process is byzantine must necessarily be private to that process.

There are several ways to model such distributed programs. One approach is to utilize message passing, where all process

information is private. And, channels are introduced between processes so that they can send information to other processes. We can observe that in such a system, a channel is a shared object that is accessed by the processes on both ends of it. To manage complexity, however, it is desirable to utilize models that hide such detailed implementation. One such approach is to use shared memory model. In shared memory model, each process can read the (public) variables of its neighbors (in a pre-defined neighbor list) and update its own information.

Modeling Distributed Programs. To capture shared memory programs, we use the approach in [12]. Specifically, we introduce w_i as a set of variables that process i can write and r_i as a set of variables process that i can read. We model these restrictions as follows:

Write Restriction. Given a transition (s_0, s_1), we can trivially identify variables written in this transition. In particular, if value of a variable in s_0 differs from that in s_1 then that variable has been written by the transition. Hence, we can model write restriction by preventing process i from utilizing transitions that write variables are outside w_i. Thus, a write restriction prevents a process from utilizing transitions:

$$write(i, w_i) = \{(s_0, s_1) | \exists x \in w_i :: x(s_0) \neq x(s_1)\}$$

To model write restrictions, we can simply add the above transitions to the transitions that violate safety as far as process i is concerned. If a transition cannot be executed by any process then we can simply add it to transitions that violate safety (while specifying input for Algorithm 1).

Read Restriction. A state s_0 is uniquely decided by knowing the values of all program variables. Hence, any transition (s_0, s_1) essentially reads all variables. Hence, modeling read restrictions requires a slightly different approach. If a process executes a transition (s_0, s_1) but it cannot read some variable, say v, then it must execute a corresponding transition in s_0' where s_0 and s_0' are identical except for the value of v. In other words, read restriction causes program transitions to be grouped.

Now, consider the case where two transitions (s_0, s_1) and (s_0', s_1') are grouped due to read restrictions. We now identify how s_0' and s_1' are related to s_0 and s_1. For simplicity of discussion, we consider the case where $w_i \subseteq r_i$, i.e., processes that can write a variable can also read it. Under these circumstances, if (s_0, s_1) and (s_0', s_1') are grouped due to read restrictions then values of all variables in r_i are equal in s_0 and s_0'. In other words, due to read restrictions, s_0 and s_0' are indistinguishable for process i. Likewise, s_1 and s_1' must also be indistinguishable for process i.

For variables not in r_i, based on our assumption that $w_i \subseteq r_i$, neither transition (s_0, s_1) nor (s_0', s_1') can change that variable. Hence, the group of transitions associated with (s_0, s_1) is given by the following formula.

$$\begin{aligned} &group(i, r_i)(s_0, s_1) = \\ &\{(s_0', s_1') | (\forall x \in r_i :: x(s_0) = x(s_0') \wedge x(s_1) = x(s_1')) \wedge \\ &\qquad (\forall x \notin r_i :: x(s_0') = x(s_1') \wedge x(s_0) = x(s_1))\} \end{aligned}$$

We handle the distribution issues during Algorithm 1 as follows. Write restrictions are added to safety violating transitions and, hence, would be removed. Read restrictions are ignored initially, i.e., in the loop 4-13. However, just before we check the condition on Line 13 that determines whether the loop should terminate, we attempt to remove the corresponding group of transitions removed in the loop. However, if removal of some transition, say t_1, causes removal of transition $t2$ and $t2$ is a transition in $I \times I$ then we only remove $t1$ but not the corresponding group. This is due to the fact that Problem III.1 prevents us from removing transitions in $I \times I$. It is therefore possible that the transitions of the graceful program may not satisfy the read restrictions entirely. Based on this discussion, we present the following algorithm snippet (Algorithm 2) that is inserted between Line 12 and Line 13 in Algorithm 1 to handle read restrictions.

Algorithm 2 Handle read restriction, insert between Line 12 and Line 13 in Algorithm 1

```
for each process j do
  p_r := p_old − p'
  p_{r_j} := transitions of process j in p_r
  for each t ∈ p_{r_j} do
    p' := p' − (group(j, r_j)(t) − I × I)
  end for
end for
```

Theorem 2. *Let 1-R denote the algorithm obtained by adding Algorithm 2 to Algorithm 1 between lines 12 and 13. Then, Algorithm 1-R solves problem* III.1 *in polynomial time in the presence of read-write restrictions.*

Proof: All the transitions that have been removed in one pass through the iteration are collected and the respective read restriction group, for each process, is removed. This group does not contain any transition $\tau \in I \times I$. Thus, problem III.1 (A1) is satisfied. Since all deadlock states in I' are removed, and *maxp* returns the largest program, problem III.1 (A2) is satisfied. Finally, problem III.1 (A3) is satisfied as no transition in I is removed and that p satisfies $spec$ from I. ■

VII. Case Study for Graceful Distributed Programs

In this section, we present the case study for byzantine agreement problem. In the canonical version of byzantine agreement, there is a general process and three non-general processes. In an ideal scenario, the general sends a decision (either 0 or 1) to non-generals. The non-generals receive this decision and finalize their decision to be that of the general. This ensures *strong validity*, i.e., the decision of the non-generals is the same as that of the general and *strong agreement*, i.e., the decision of the non-generals match with each other.

One can consider some graceful versions of this. One graceful version allows a non-general to be byzantine and, hence, allows it to change its decision. For this graceful version, the program satisfies *weak validity*, i.e., the decision of the non-byzantine non-general is the same as that of the general provided the non-general is non-byzantine and *weak agreement*, the decision of the non-general is the same as

that of another non-general provided both are not byzantine. [2] The program for the ideal scenario suffices to deal with ensuring weak validity and weak agreement if a non-general is byzantine.

In this case study, we begin with this program as our input program for which we want to add graceful degradation. [3] Subsequently, we derive a graceful version of this program that only satisfies `weak agreement`. Finally, we utilize this derived graceful program along with the original program to derive a fault-tolerant graceful degradation byzantine agreement program, i.e., a program that (1) satisfies `weak validity` and `weak agreement` in the absence of faults and (2) satisfies `weak agreement` in the presence of faults, where the fault makes the general process byzantine.

Original Program.

1) State Space: The program consists of one *general* process (g) and three *non-general* processes, i, j and k. Each process maintains a decision variable d. For process g, the value for decision d can either be 0 or 1. For processes i, j and k, the decision value can be $\bot, 0$ or 1. The value $\bot$ denotes that the corresponding non-general process has not received the decision from general. Each non-general also maintains a variable f that is 0 initially and is set to 1 when it finalizes its decision. Also, the variable b maintained in each process denotes whether the process is byzantine ($b = 1$) or not ($b = 0$). Thus, the variables are defined as follows.

- $d.g : \{0, 1\}$, $d.i, d.j, d.k : \{0, 1, \bot\}$, $b.g, b.i, b.j, b.k : \{true, false\}$
- $f.i, f.j, f.k : \{0, 1\}$

2) Specification of Byzantine Agreement: As described above, the original safety specification is `weak validity` and `weak agreement`. Weak validity requires non-byzantine non-generals to finalize their decision to be the same as that of the general. Weak agreement requires that the final decision of any two non-byzantine non-generals should be the same. Additionally, the notion of finalization requires that a non-general cannot finalize the decision $\bot$ and it cannot change it after it finalizes it. Thus, the set of bad states or transitions identified by these specifications are as follows:

$$
\begin{aligned}
spec_{bs\_va} =& \exists l \in i, j, k :: \\
& \neg b.l \wedge d.l \neq d.g \wedge f.l = 1 \\
spec_{bs\_ag} =& \exists p, q \in i, j, k :: \\
& \neg b.p \wedge \neg b.q \\
& \wedge d.p \neq d.q \\
& \wedge f.p = 1 \wedge f.q = 1 \\
spec_{bs\_nb} =& \exists l \in i, j, k :: \\
& \neg b.l \wedge f.l = 1 \wedge d.l = \bot \\
spec_{bt\_fi} =& \{(s, s') | \exists l \in i, j, k :: \\
& \qquad s(f.l) = 1 \wedge s(b.l) = 0 \wedge \\
& \qquad (s(d.l) \neq s'(d.l) \vee s'(f.l) = 0)\}
\end{aligned}
$$

[2]The Agreement requirement generally considered in the literature corresponds to `weak agreement` from this paper. The Validity requirement generally considered in the literature is slightly different from `weak validity` considered here. Specifically, `weak validity` requires the non-general to be non-byzantine. But does not impose the same requirement on the general. This is due to the fact that `weak validity` is expected to be satisfied in the absence of faults (that make the general byzantine) and only `weak agreement` is expected to be satisfied in the presence of faults.

[3]We could also apply Algorithm 1 to the program where the original specification is `strong validity` and `strong agreement`. However, the corresponding derivation is outside the scope of this paper.

The original safety specification, $spec$, is specified in terms of the set of bad states, $(spec_{bs\_va} \cup spec_{bs\_ag} \cup spec_{bs\_nb})$, and the set of bad transitions $spec_{bt\_fi}$.

Finally, the liveness specification, Lv requires (for both original and graceful program) that each non-byzantine non-general finalizes its decision. This is specified as follows:

$Lv := true \rightsquigarrow (\forall p \in i, j, k :: (b.p = 0) \rightarrow (f.p = 1))$

3) Actions for Original Program: As discussed above, the original program deals with the case where a non-general is byzantine but not the case where a general is byzantine. The actions of the original program are as follows:

$d.j = \bot \wedge f.j = 0 \longrightarrow d.j := d.g$
$d.j \neq \bot \wedge f.j = 0 \longrightarrow f.j := 1$
$b.j \longrightarrow d.j := 0|1$

The last action is the one that allows a byzantine non-general to change its decision. Since Problem Statement III.1 guarantees to preserve existing behavior, these actions would be preserved in the final program. These actions are treated as environment actions, i.e., actions that eventually stop. Hence, they are not considered in resolving liveness on Lines 11 and 12.

4) Invariant of Original Program: The invariant can be computed from the initial state where the general is not byzantine and at most one non-general may be byzantine. Moreover, the decision of each non-general is $\bot$ and it has not finalized its decision. The states reached from this initial state by the computation of the above program are as follows:

$$
\begin{aligned}
I = & \neg b.g \wedge (\neg b.i \vee \neg b.j) \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.i \vee \neg b.k) \\
& \wedge (\forall p :: \neg b.p \Rightarrow (d.p = \bot \vee d.p = d.g)) \\
& \wedge (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \bot))
\end{aligned}
$$

Graceful program. The graceful program is intended for the scenarios where the general is byzantine. Hence, the weaker specification $spec_r$ is specified in terms of the set of bad states, $(spec_{bs\_ag} \cup spec_{bs\_nb})$, and the set of bad transitions $spec_{bt\_fi}$.

We now illustrate Algorithm 1 in the context of byzantine agreement. In particular, we instantiate Algorithm 1 with (1) p, the transitions of the original program, (2) $spec$ and $spec_r$ representing the original and weaker specification, (3) liveness specification Lv, (4) invariant I, and (5) state predicate S_a equal to $b.g \wedge \neg b.i \wedge \neg b.j \wedge \neg b.k$. Note that the last parameter is based on the observation that the graceful program is intended for scenarios where the general is byzantine. Now, we evaluate Algorithm 1 on these inputs.

Lines 1-2: Identifying invariant (I'). The algorithm starts with generating all possible states in invariant satisfying weaker safety specification. Since we include original invariant (I) in the fault-intolerant program, the invariant (I_Δ) not in I is enumerated as follows.

1) First, observe that S_a requires that only the general is byzantine and no non-general is byzantine.
2) Considering states in S_a further, we observe (a) I_Δ includes all states where $d.i = d.j = d.k$ is true. This is due to the fact that in such states agreement is satisfied irrespective of values of $b.i, b.j, b.k, b.g., f.i, f.j, f.k$ or $d.g$. (b) Now, consider states where some two non-generals, say i and j differ. To ensure `Agreement`, in such states in I_Δ, either $d.i$ (or $d.j$) should be equal

to $\bot$ or $f.i$ (or $f.j$) must be 0. (c) If I_Δ includes a state where $f.j$ is 1 then $d.j$ must be different from $\bot$.

Hence, after computation by Line 1, we have

$$I_\Delta = (b.g = 1 \wedge b.i = b.j = b.k = 0) \wedge (\forall l \in i,j,k : f.l = 1 \rightarrow d.l \neq \bot) \wedge \bigvee \left(\begin{array}{l} d.i = d.j = d.k \\ d.i \neq d.j \rightarrow (d.i = \bot \vee d.j = \bot \vee f.i = 0 \vee f.j = 0) \\ d.i \neq d.k \rightarrow (d.i = \bot \vee d.k = \bot \vee f.i = 0 \vee f.k = 0) \\ d.j \neq d.k \rightarrow (d.j = \bot \vee d.k = \bot \vee f.j = 0 \vee f.k = 0) \end{array} \right)$$

Line 3: Identifying a set of transitions (p') not violating safety specification. After enumerating the possible states in I_Δ generated on Line 1, the algorithm generates p' by reusing p as well as including any transition in space $I_\Delta \times I'$ that does not violate the specification $spec_r$.

If following Algorithm 1 as is, it would include transitions where process j changes the value of $d.k$. Or, it may include transitions of process j that rely on the general being non-byzantine. Since this would be unacceptable in the fault-tolerant graceful degradation program, as it would be impossible to implement such a program, we utilize the approach discussed in Section VI to model ability of different processes to read and write variables.

Loop 4-13: Resolving deadlock states and liveness violation. It is straightforward to observe that in this case, the following transitions can be included in p': (1) A process that has not finalized can change its decision to 0 or 1 non-deterministically, or (2) A process can finalize as long as its decision is not $\bot$ and another process has not finalized with a different decision.

Notation. We use the sequence $\langle x_1, x_2, x_3, x_4 \rangle$ to denote the set of states where the value of x_1 equals $d.g$, the value of x_2 equals $d.i$, the value of x_3 equals $d.j$ and the value of x_4 equals $d.k$. We use '$*$' to denote that the decision value of certain process is irrelevant. For example $\langle *, 1, 1, 1 \rangle$ denotes the set of states where $d.i = d.j = d.k = 1$ and $d.g$ is either 0 or 1.

In such a program, there are no deadlock states. However, some transitions need to be removed due to non-decreasing rank value (Line 11). In particular, consider a state in $\langle *, 0, 0, 1 \rangle$ where process k has not finalized its decision. Further consider the transition where process i changes its decision to 1. This transition is removed since the rank of both states (number of steps to reach a state where all non-byzantine non-generals have finalized) is the same. In other words, a process in majority cannot change its decision to the minority process unless that minority process has finalized its decision.

At this point, we reach the end of the loop (Line 13). Since we need to consider grouping caused by read restrictions, this results in removal of all transitions where a process in majority changes its decision to one that is in the minority, irrespective of whether the minority process has finalized. (Note that this removal does not impact transitions in I and the invariant of the original program, since there is no corresponding state in I.)

While we omit the detailed analysis of different iterations of this loop, we note that the transitions of the graceful program for process i in states outside I are as shown in Figure 6. (Recall that the transitions for process i inside I are the same as that of the original program.):

$d.i = \bot$	$\longrightarrow d.i := d.g$
$d.i = \bot \wedge (d.j = d.k = \bot)$	$\longrightarrow d.i := 0 \vert 1$
$d.i = \bot \wedge f.i = 0 \wedge ((d.j \neq d.g \wedge d.j \neq \bot) \vee (d.j \neq d.k \wedge d.k \neq \bot))$	$\longrightarrow d.i := 0 \vert 1$
$d.i \neq \bot \wedge f.i = 0 \wedge (d.i = d.j \vee d.i = d.k)$	$\longrightarrow f.i := 1$
$d.i \neq \bot \wedge f.i = 0 \wedge (d.j = d.k = \bot \wedge d.i = d.g)$	$\longrightarrow f.i := 1$
$d.i \neq \bot \wedge f.i = 0 \wedge$ $d.j \neq \bot \wedge d.i \neq d.j \wedge d.j = d.k$	$\longrightarrow d.i := d.j$

Fig. 6: Byzantine Agreement: Actions for Graceful Program

VIII. Adding Fault-tolerance to Program with Graceful Degradation

In this section, we present the second step in the two-step approach, namely, adding fault-tolerance to the graceful program. Hence, we first introduce the notion of faults. Then, in Problem Statement VIII.2, we formalize the requirements for adding fault-tolerant graceful-degradation. Subsequently, we continue with the two case studies discussed in Section V by utilizing the original and graceful program to obtain a fault-tolerant graceful-degradation program. For brevity, we only focus on one of the levels of tolerance, namely masking tolerance.

The faults, say f, that a program is subject to is represented by a set of transitions. Specifically, given a program $p = \langle S_p, \delta_p \rangle$, faults f are a subset of $S_p \times S_p$. We use $p[]f$ to denote the transitions obtained by taking the union of p and f.

Masking fault-tolerance. One can consider different levels of tolerance, namely failsafe, nonmasking and masking, to a given fault based on whether the program satisfies safety and/or liveness in the presence of faults. Masking fault-tolerant program ensures that in the absence of faults, it satisfies its specification (including both safety and liveness). Moreover, in the absence of faults, it remains inside its invariant. In the presence of faults, it may be perturbed to a state outside its invariant. Let T be the boundary up to which the program can be perturbed due to faults and subsequent program actions. A masking fault-tolerant program ensures that starting from any state in T, it recovers to its invariant. Thus, it ensures that after faults occur, the program recovers to states from where both safety and liveness are satisfied. Additionally, during this recovery, it satisfies the safety specification.

Based on this intuition, we formally define a program, say p, is masking f-tolerant to $spec$ (=$\langle Sf, Lv \rangle$) from I iff the following conditions hold:

(1) p satisfies $spec$ from I.

(2) $\exists T$ :: (a) $I \subseteq T$. (b) $p[]f$ satisfies Sf from T. (c) Every computation of $p[]f$ that starts from a state in T has a state in I.

Now we consider the problem of adding fault-tolerance to the graceful program. Intuitively, the resulting fault-tolerant

program, say p_f, is required to satisfy the original (stronger) specification in the absence of faults. However, when the program is perturbed by faults, the program is guaranteed to recover to states from where it satisfies the weaker specification. We formally identify the problem in Problem Statement VIII.2, as follows.

Problem Statement VIII.2:
Addition of Fault-Tolerant Graceful-Degradation
Given p, $spec$, I, such that p satisfies $spec$ from I;
p_g, $spec_r$, I_g, such that p_g satisfies $spec_r$ from I_g;
$I \subseteq I_g$, $p \mid I \subseteq p_g \mid I$, f.
Does there exist I_f, I_{g_f} and p_f such that
B1:$I_f \subseteq I$, $p_f \mid I_f \subseteq p_g \mid I_{g_f}$
B2:$I_{g_f} \subseteq I_g$, $p_f \mid I_{g_f} \subseteq p_g \mid I_{g_f}$.
B3:p_f satisfies $spec$ from I_f.
B4:p_f is masking f-tolerant for $spec_r$ from I_{g_f}.

As one can imagine, it should be possible to reuse existing algorithms for adding fault-tolerance to add fault-tolerant graceful degradation. However, one needs to ensure that during such reuse, the synthesized program satisfies the weaker specification in the presence of faults. By contrast, (assuming that satisfying the stronger specification were not feasible), the existing algorithm for adding fault-tolerance will declare failure to add fault-tolerance. There are several algorithms [1], [5], [13] for adding fault-tolerance in the literature. In this section, we plan to utilize them as a black box, i.e., we only rely on the assumption that they satisfy the problem of adding fault-tolerance (repeated from [5]).

Problem Statement VIII.3:
Addition of Fault-Tolerance
Given p, $spec$, I, f such that p satisfies $spec$ from I;
Does there exist I_f, and p_f such that
C1:$I_f \subseteq I$, $p_f \mid I_f \subseteq p \mid I_f$
C2:p_f is masking f-tolerant for $spec$ from I_f.

In order to describe the fault-tolerant program subject to the constraints in Problem Statement VIII.2, we can utilize any of the algorithms [1], [5], [13]. We use the name $Add\_masking$ to describe such a generic algorithm. Since our reuse is black-box in nature, we only rely on the proof (from [1], [5], [13]) that it satisfies Problem VIII.3. However, for the convenience of the reader, we briefly describe the key steps of these algorithms. Given the fault-intolerant program, a set of fault transitions, the invariant and specification, $Add\_masking$ first ensures all the reachable states by $p[]f$ satisfy safety specification and then excludes the computation that starts outside the invariant and cannot possibly reach a state in the invariant. In addition, $Add\_masking$ also resolves deadlock states and removes computations that reach the deadlock states if it is impossible to add recovery.

The algorithm for adding fault-tolerant graceful degradation is as shown in Algorithm 3. First, it invokes $Add\_masking$ (Line 4) that satisfies the constraints of Problem VIII.3. The input to $Add\_masking$ consists of the graceful program p_g, fault transition f, invariant of the graceful program I_g, and weaker specification $spec_r$. $Add\_masking$ returns p' and I' such that constraints of Problem VIII.3 are satisfied. Note that p' satisfies the weaker specification in the absence of faults. Hence, we consider the computations of p' on $I \cap I'$ (Line 5) by computing $I \cap I'$ and $p''|(I \cap I')$, i.e., a program whose transitions are a subset of the transitions of p and p'. Since the transitions of this program are a subset of that of p, it satisfies the stronger specification, $spec$, as long as it does not deadlock in any state in $I \cap I'$. If there are deadlock states, those are removed (Loop 6-11) and the process is repeated (Loop 2-12).

Algorithm 3 Adding Fault-tolerance to Graceful Program

Input: fault-intolerant program p, graceful program transitions p_g, fault transitions f, invariant I, graceful invariant I_g, weaker specification $spec_r$.
Output: masking fault-tolerant program p_f
1: $p' := p_g$, $I' := I_g$, $I'' := I$
2: **repeat**
3: $I_{old} := I'$, $p_{old} := p'$
4: $p', I' := Add\_masking(p', f, I', spec_r)$
5: $I'' := I'' \cap I'$, $p' := p' - (I'' \times \neg I'')$, $p'' := p'|I''$
6: **while** $deadlock(I'', p'') \neq \emptyset$ **do**
7: $I' := I' - deadlock(I'', p'')$
8: $I'' := I'' \cap I'$
9: $p' := maxp(p', I'', I')$
10: $p'' := p'|I''$
11: **end while**
12: **until** $I_{old} = I' \wedge p_{old} = p'$
13: return p' as p_f, I' as I_{g_f}, and I'' as I_f

We now prove the correctness of Algorithm 3.

Lemma 3 (Correctness of Algorithm 3)**.** *Let the input to Algorithm 3 be:*

- *fault-intolerant program p,*
- *invariant I,*
- *spec (original specification),*
- *graceful program p_g,*
- *$spec_r$ (relaxed specification),*
- *I_g (graceful invariant), and*
- *fault f*

Let the output of Algorithm 1 be:

- *fault-tolerant graceful program p_f,*
- *stronger invariant I_f (in the absence of f), and*
- *invariant I_{gf}*

Then

1) $I'' \subseteq I$
2) $I' \subseteq I_g$
3) p' *satisfies* spec *from I"*
4) p' *is masking f-tolerant for $spec_r$ from I'*

Proof: The proof is on a case basis.

- $I'' \subseteq I, p' \mid I'' \subseteq p_g \mid I'$: This follows from Lines 1, 5, 7-8, as I'' can only become smaller than I. Also, since $p' \subseteq p_g$ and $I'' \subseteq I'$ (Lines 7 and 8), we have that $p' \mid I'' \subseteq p_g \mid I'$.
- $I' \subseteq I_g, p' \mid I' \subseteq p_g \mid I'$: This follows from the fact that $I' = I_g$ initially and from the properties of *Add_masking* (Lines 4 and 7).
- p' satisfies *spec* from I'': It follows from the fact that I'' is closed in p' (Line 5), $I'' \subseteq I$ (1st condition) and p satisfies $spec$ from I (from problem statement assumption).

– p' is masking f-tolerant for $spec_r$ from I': This follows from the properties of *Add_masking* (Line 4) and Line 7 (as I' gets stronger).

■

Lemma 4. *Let the input to Algorithm 3 be:*
– *fault-intolerant program p,*
– *invariant I,*
– *spec (original specification),*
– *graceful program p_g,*
– *$spec_r$ (relaxed specification),*
– *I_g (graceful invariant), and*
– *fault f*

Then, algorithm 3 terminates in polynomial time in state space of p.

Proof: We need to show that the two loops eventually terminate. We prove this by showing the existence of a least fixpoint for both loops.

– **Inner loop**: We need to show that, for a given I'' and p'', eventually *deadlock*$(I'', p'') = \emptyset$. When *deadlock*$(I'', p'') \neq \emptyset$, then I'' and p'' become smaller (Lines $7-10$). Eventually, $I'' = \emptyset$, making $p'' = \emptyset$. At the next iteration, *deadlock*$(I'', p'') = \emptyset$ and the loop terminates.
– **Outer loop**: If the inner loop terminates with $I'', p'' = \emptyset, \emptyset$, then the outer loop will terminate in the next iteration, since $I_{old}, p_{old} = \emptyset, \emptyset$, which is equal to the terminating condition of the outer loop. On the other hand, if the inner loop terminates with $I'', p'' \neq \emptyset, \emptyset$, I_{old} and p_{old} get smaller in the next iteration (Line 3). Eventually, $I_{old}, p_{old} = \emptyset, \emptyset$ and the loop terminates.

■

Theorem 3. *Let the input to Algorithm 3 be: (i) fault-intolerant program p, (ii) invariant I, (iii) spec (original specification), (iv) graceful program p_g, (v) $spec_r$ (relaxed specification), (vi) I_g (graceful invariant), and (vii) fault f*
Then, Algorithm 3 solves Problem VIII.2 in polynomial time.

Proof: It follows from Lemmas 3 and 4, $p_f \mid I_f \subseteq p_g \mid I_{gf}$ (since $I_f \subseteq I_{gf}$ and p_f is obtained by removing transitions from p_g) and $p_f \mid I_{gf} \subseteq p_g \mid I_{gf}$ (as p_f is smaller than p_g). ■

A. Case Study (continued): Fault-tolerant Printer System

In this section, we continue with the printer system from Section V. We consider the fault that dequeues the next task before the current task is printed. Specifically, the fault action is represented as follows:

$f :: d_i = 1 \wedge p_i = 0 \longrightarrow d_{i+1} := 1$

Next, we use this fault action along with the original and graceful program from Section V to generate the fault-tolerant graceful degradation program: Observe that in this case, the invariant of the graceful program is closed in the fault actions. Hence, *Add_masking* can trivially satisfy the fault-tolerance requirement as no recovery action is needed. And, the actions of the fault-tolerant graceful-degradation program are the same as those of the graceful program.

B. Case Study (continued): Resource Constraint Problem

In this section, we extend the cellular network case study from Section V to make it fault-tolerant. One type of fault that occurs often is channel loss. The fault action is:

$f :: av > 0 \longrightarrow av := av - 1$

As can be observed, the invariant of the graceful program is closed in the fault action, meaning that no further action needs to be added to the graceful program, i.e., the graceful program can tolerate channel losses.

C. Case Study (continued): Fault-tolerant Byzantine Agreement

As discussed earlier, the fault action relevant for this step is the one that makes the general process byzantine. We represent this fault by two actions. First action causes the general to be byzantine and the second allows it to change its decision.

$f :: \quad \forall p \in g, i, j, k \ \neg b.p \longrightarrow b.g := true$
$\quad\quad b.g \longrightarrow d.g := 0|1$

Based on the constraints of Problem VIII.2, in the context of byzantine agreement, the problem of adding fault-tolerant graceful-degradation will result in a program that has the following properties: In the absence of faults, i.e., when the general is not byzantine, it will guarantee that `weak validity`, `weak agreement` and Lv are satisfied. Moreover, in the presence of faults, i.e., when the general is byzantine, `weak agreement` will be satisfied and eventually the program will recover to a state where both `weak agreement` and Lv would be satisfied. Observe that taken together, this ensures that the final program guarantees that if the general is not byzantine then all non-generals finalize with a decision that is the same as that of the general. And, if the general is byzantine, all non-byzantine non-generals finalize with identical decision. In other words, the fault-tolerant graceful degradation program satisfies the typical requirements for byzantine agreement [14].

The graceful program from Section VII did not entirely handle the read restrictions. Hence, the input program would remove certain transitions that potentially violate the read restrictions. Observe that the first action (where process i changes from $\perp$ to $d.g$) exists in I as well as outside I. Hence, knowledge of $b.g$ is not needed when executing this action. Hence, the action corresponding to the group containing second action must be removed. Likewise, in the third action, i can change its value to either 0 or 1 only if both $d.j$ and $d.k$ are different from the general. The fourth action remains as is. Regarding the fifth action, we observe that if a fault occurs in a state reached after executing that action then the corresponding state, where $d.g \neq d.i$ and $f.i = 1$, is outside I_g. Since recovery is not possible from this state, this transition would be removed during addition of fault-tolerance. Finally, the sixth action remains as is. Thus, the actions of the fault-tolerant program are as shown in Figure 7. And as discussed above, it satisfies the requirements of [14]. The program is also correct even if the third modified action is removed. It is generated by our program to provide maximal choice to the designer.

$d.i = \bot \longrightarrow d.i := d.g$

Second action subsumed by the first one.

Third action modified as below

$d.i = \bot \wedge f.i = 0 \wedge (\ (d.j \neq d.g \wedge d.j \neq \bot) \wedge (d.j \neq d.k \wedge d.k \neq \bot)) \longrightarrow d.i := 0|1$

$d.i \neq \bot \wedge f.i = 0 \wedge (d.i = d.j \vee d.i = d.k) \longrightarrow f.i := 1$

Fifth action removed

$d.i \neq \bot \wedge f.i = 0 \wedge$
$d.j \neq \bot \wedge d.i \neq d.j \wedge d.j = d.k \longrightarrow d.i := d.j$

Fig. 7: Byzantine Agreement: Actions for Fault-tolerant Graceful-Degradation Program

IX. Addition of Fault-Tolerant Multi-Graceful Degradation

The problem of graceful degradation focuses on obtaining a program that satisfies a weakened specification. This concept can be generalized to a hierarchy of specifications, one weaker than the previous one. In this section, we generalize our definitions and algorithms to such multi-graceful degradation.

A. Problem Statement

A multi-graceful degradation program satisfies its original specification when no fault has occurred. However, if faults occur then depending upon their severity, it satisfies a weaker specification. To characterize the severity of faults, a multi-graceful degradation program identifies a set of faults, say $f_1, f_2, \cdots, f_n$, such that $\forall j : 0 < j < n : f_j \subseteq f_{j+1}$. Since $f_j \subseteq f_{j+1}$, f_{j+1} can perturb the program at least as much as f_j does. Hence, we say that f_{j+1} is *more severe* than f_j. In the presence of increasingly severe faults, multi-graceful degradation program provides successively weaker guarantees. Hence, it identifies a set of specifications $spec_{r_0} (= spec$, the original specification), $spec_{r_1}, spec_{r_2}, \cdots, spec_{r_n}$ such that $\forall j : 0 \leq j < n : spec_{r_{j+1}}$ is weaker than $spec_{r_j}$. Moreover, in the presence of increasingly severe faults, the program may not recover to its original behavior. Hence, it identifies a set of state predicates $I'_1, I'_2, \cdots I'_n$ to which it recovers after the recovery is complete. For each class of faults, a fault-tolerant multi-graceful degradation program provides the desired level –masking, failsafe or nonmasking– of tolerance. However, in this section, we consider masking fault-tolerance alone. Generalization to other types of fault-tolerance is straightforward.

To add multi-graceful degradation to a given program p that satisfies its specification, say $spec$ from invariant, say I, we need to construct a program p_f with invariant I_f such that p_f satisfies $spec$ from I_f without using any new behaviors other than those used by p. However, if faults occur then it needs to provide appropriate level of tolerance to them as well as recover to states from where it satisfies the corresponding weaker specification. Thus, the problem statement is as follows:

Problem Statement: IX.1
Addition of Fault-Tolerant Multi Graceful-Degradation
Given
∘ p, $spec$, I, such that p satisfies $spec$ (= $spec_{r_0}$) from I,
∘ set of faults, $f_1, f_2, \cdots, f_n$, where
$\forall j : 0 < j < n : f_j \subseteq f_{j+1}$,
∘ set of relaxed specifications $spec_{r_1}, spec_{r_2}, \cdots, spec_{r_n}$, where $\forall j : 0 \leq j < n : spec_{r_{j+1}}$ is weaker than $spec_{r_j}$

Generate a multi-graceful fault-tolerant program p_f and invariant I_f such that
$\exists I'_1, I'_2, \cdots I'_n$ where $I_f \subseteq I'_1 \subseteq I'_2 \subseteq \cdots \subseteq I'_n$
1: $I_f \subseteq I$, $p_f|I_f \subseteq p|I_f$
2: p_f satisfies $spec$ from I_f
3: p_f is masking f_i-tolerant to $spec_{r_i}$ from I'_i.

B. Algorithm for Adding Fault-Tolerant multi-graceful degradation

In this section, we introduce the algorithm for adding multi-graceful degradation in Algorithm 4. Algorithm 4 takes as input the set of increasingly severe faults and correspondingly weaker specifications. It also takes in as inputs state predicates $S_{a_1}, S_{a_2}, \cdots, S_{a_n}$. These state predicates capture constraints, if any, under which the corresponding graceful behavior is expected. By default, they could be set to $S_p - I$, i.e., all states outside the invariant of the original program.

Algorithm 4 considers the faults successively. For fault class f_1, it generates a graceful program satisfying a relaxed specification $spec_{r_1}$ (Line 3). This program preserves all original behaviors of p and adds new behaviors that satisfy $spec_{r_1}$. Observe that this can be achieved by Algorithm 1 presented earlier in the paper. After obtaining the graceful program that satisfies $spec_{r_1}$, we add fault tolerance to f_1 (Line 4). This algorithm adds masking tolerance to the input program. Similar to Algorithm 3, we use a generic algorithm $Add\_masking$ to achieve this. This algorithm can be instantiated to be an algorithm from [1], [5], [13].

Finally, this whole process is repeated for each fault-class. Thus, the algorithm for adding multi-graceful degradation is as shown in Algorithm 4:

Theorem 4 (Correctness of Algorithm 4)**.** *Let the input to Algorithm 4 be:*

- *fault-intolerant program p,*
- *invariant I,*
- *state predicates $S_{a_1}, S_{a_2}, \ldots, S_{a_n}$,*
- *$spec_{r_1}$, $spec_{r_2}$, …, $spec_{r_n}$*
- *fault transitions $f_1, f_2 \ldots f_n$,*

Let the output of Algorithm 4 be:

- *masking fault-tolerant graceful program p_f,*
- *invariant I_f*

Then

1) *$I_f \subseteq I$, $p_f|I_f \subseteq p|I_f$.*
2) *p_f satisfies $spec$ from I_f.*
3) *p_f is masking f_i-tolerant to $spec_{r_i}$ from I_i.*

Algorithm 4 Adding multi-graceful degradation

Input: fault-intolerant program p, fault-intolerant program invariant I.
state predicates $S_{a_1}, S_{a_2}, \cdots, S_{a_n}$
weaker specifications $spec_{r_1}$, $spec_{r_2}$, $\cdots$, $spec_{r_n}$.
fault transitions $f_1, f_2, \cdots, f_n$.
Output: masking fault-tolerant program p_f, and its invariant I_f

1: $p'' := p, I'' := I$
2: **for** $i = 1$ to n **do**
3: p_{gi}, I_{gi} := Gen_graceful(S_{ai},p'',I'' ,$spec_{r_i}$) // This function invokes Algorithm 1 .
4: p', I' :=Add_masking(p_{gi},f_i, I_{gi}, $spec_{r_i}$) // If any of the above two procedures fail, then algorithm aborts and reports no result returned.
5: $p'' := p'|I''$, $I'' := I'' \cap I'$
6: **while** $deadlock(I'', p'') \neq \emptyset$ **do**
7: $I' := I' - deadlock(I'', p'')$
8: $I'' := I'' \cap I'$
9: $p' := maxp(p', I'', I')$
10: $p'' := p'|I''$
11: **end while**
12: $p'' := p'$, $I'' := I'$
13: **end for**
14: return p'' as p_f and $I'' \cap I$ as I_f

Function: $maxp$(t: transition predicate, S_1,S_2,$\cdots$:set of state predicates where $S_1 \subseteq S_2 \subseteq \cdots$)
return $\{(s_0, s_1)|t \cap (s_0 \in S_1 \Rightarrow s_1 \in S_1) \wedge (s_0 \in S_2 \Rightarrow s_1 \in S_2) \wedge \cdots\}$

Function: $deadlock$(S: state predicate , t: transition predicate)
return $\{s_0|s_0 \in S \wedge (\forall s_1 \in S : (s_0, s_1) \notin t)\}$

Proof:

To show Algorithm 4 is sound, we first recall the properties of Algorithm 1. Let the input to Algorithm 1 be p, its invariant I, original specification $spec$ and the weaker specification $spec_r$,. Let the output be the graceful program p_g and its invariant I_g. Then, we have:

- $p_g|I = p|I$, $I \subseteq I_g$.
- p_g satisfies $spec_r$ from I_g.
- p_g satisfies $spec$ from I.

Now we use the above three properties to prove this theorem where we need to show that p_f and I_f satisfy conditions in Problem Statement IX.1.

- $I_f \subseteq I$, $p_f|I_f \subseteq p|I_f$.
  $I_f \subseteq I$ follows from Line 14. $p_f|I_f \subseteq p|I_f$ follows from Line 4 since it does not add any new transition to $p|I_f$.
- p_f satisfies $spec$ from I_f.
  Note that after each iteration (Line 2 to Line 13), p'' satisfies the specification from invariant generated in previous steps. It follows from Line 3 that I'' is closed in p_{g_i}; and from Line 4 that I'' is closed in p' since $Add\_masking$ does not add any new transition in I_{g_i} as well as in I''. In addition, from Line 6 to Line 11, no new transition is added and deadlock states are resolved. Hence I_f is closed in p_f. Moreover p_f satisfies $spec$ since $I_f \subseteq I$.
- p_f is masking f_i-tolerant to $spec_{r_i}$ from I_i.
  p'' obtained after each iteration (Line 2 to Line 13) satisfies $spec_{r_i}$ follows the same proof above. And p_f is masking f_i-tolerant is ensured via $Add\_masking$.

■

Theorem 5. *Algorithm 4 is polynomial in the size of* $(|S_p| * n)$ *where* S_p *is the state space and* n *the is number of fault classes.*

Proof: The complexity of Algorithm 1 is polynomial in the size of $(|S_p|)$. If we utilize the $Add\_masking$ algorithm from [5] then its complexity is also polynomial in the size of $(|S_p|)$. Moreover, in Algorithm 4, the outer loop (from Line 2 to Line 13) executes at most n iterations and the inner loop (from Line 6 to Line 11) executes polynomial times in $(|S_p|)$. Therefore, the complexity of Algorithm 4 is polynomial in the size of $(|S_p| * n)$. ■

X. Ohio Coal Research Center Ventilation System

In this section, we illustrate Algorithm 4 with a case study related to Ohio Coal Research Center (OCRC) ventilation system. The OCRC safety system [15] was designed to protect personnel and facilities by detecting the presence of explosive and/or toxic gas and either dealing with it, or by alerting lab personnel and emergency responders. Our goal in this paper is to illustrate how (a slightly abstracted version of) this system can be designed using Algorithm 4.

In OCRC, the facility works with several gases that are potentially explosive/poisonous in large enough concentration.

The first line of defense if these gases are released is ventilation, i.e., to evacuate any gas before it becomes a hazard. Ordinarily, ventilation is handled by the building's central air supply (see Fig 8) and exhaust, augmented by an exhaust booster fan. If gas is detected (above a specified threshold), the lab needs to be isolated from the rest of the building by closing the exhaust vent cover and then turning off the booster, and turning on a local exhaust fan that evacuates air directly outside the building. Depending on the level of gas concentration, status lights in the lab will change from red to yellow or red, and a claxon alarm will sound. In addition, automated phone calls are placed to lab personnel and emails are sent.

The system needs to tolerate equipment faults. Sensors are generally duplicated so that failure of one sensor is tolerated, and relative accuracy can be compared. Hence, failure to detect the gas is not a concern. However, other parts of the system can fail. For example, if the central exhaust supply fails, a local supply fan needs to be turned on. Also, whenever feasible, isolation is desired so that the released gas does not enter the rest of the building. However, if the vent cover fails and does not close then it may be impossible.

Based on the analysis of these faults, in [15], authors have identified four possible statuses: *Safe*, *NotLocal*, *UnsafeLab* and *UnsafeCentral*.

Status *Safe* corresponds to the case where there is ventilation, and the lab is isolated if necessary. Ventilation can be Central (central exhaust on, vent cover open, booster on) or Local (vent cover closed and local exhaust on); in both cases central and/or local supply are on. *NotLocal* corresponds to the case where there is ventilation but not isolation; this means central exhaust is being used. *UnsafeLab* corresponds to the case where the lab is isolated although ventilation is inadequate. *UnsafeCentral* corresponds to the case where the lab should be isolated but is not, and ventilation is inadequate. These form a total order from safest to most hazardous.

To illustrate graceful degradation, we focus on the key components of ventilation: central supply, central exhaust, vent cover, local supply and local exhaust. For simplicity, we ignore the exhaust booster. Specifically, if exhaust booster fails, we model it as a failure of the central exhaust system. Central supply and exhaust are not controllable but other components are. Controllable components have an actuation status (open/closed, on/off) and controllability status (controllable or not). A component is declared uncontrollable if an attempt to actuate it fails, and it remains uncontrollable until an operator fixes it and declares it controllable. Controllability lets us know whether it can be actuated or whether we have to adapt to its present status.

A. Modeling Ventilation System

Variables.

Based on the assumption made by system designers, either central supply or local supply is functioning and it is acceptable to use either one or both. Moreover, the choice of supply does not affect the status (Safe, NotLocal, etc.) of the system. Hence, we only model the central exhaust and local exhaust system. Hence, we introduce the variable cc (denoting whether central exhaust is available to use, i.e., it has not been turned off by building.) and variable lc (denoting whether local exhaust is available to use, i.e., whether it is controllable). We also introduce variable c (denoting whether the central exhaust system is being used by the lab) and variable l (denoting whether the local exhaust system is being used by the lab). Likewise, we introduce variables v and vc to denote status of vent cover and its controllability. Finally, the variable gas denotes whether hazardous gas exists at a sufficiently high concentration. Thus, the variables are as follows:

- $gas : \{0, 1\}$: Gas hazard detected ($gas = 1$) or not ($gas = 0$).
- $vc : \{0, 1\}$: Vent controllable ($vc = 1$) or not ($vc = 0$).
- $v : \{0, 1\}$: Vent closed ($v = 0$) or open ($v = 1$).
- $cc : \{0, 1\}$: Central exhaust controllable ($cc = 1$) or not ($cc = 0$).
- $c : \{0, 1\}$: Central exhaust in use ($c = 1$) or not ($c = 0$).
- $lc : \{0, 1\}$: Local exhaust controllable ($lc = 1$) or not ($lc = 0$).
- $l : \{0, 1\}$: Local exhaust in use ($l = 1$) or not ($l = 0$).

Thus, the state space of this program consists of all states obtained by assigning each variable value from the domain. Additionally, when the context is clear, we would use C-style syntax, that is $\neg v$ evaluating to $true$ if $v = 0$.

Based on the above modeling, we identify certain structural constraints. These constraints describe limitations in terms of transitions that a revised program can include. For example, if the central exhaust is not controllable then the system cannot use it by setting $c = 1$. Observe that this structural constraint can be modeled as a set of transitions that the program cannot include. Hence, during the synthesis algorithm, we model it as (an additional) safety specification that has to be satisfied by the revised program. The set of (bad) transitions identifying the structural constraints are given by the following set of transitions.

$$Sf_{bt} = \{(s_0, s_1) | (gas(s_0) \neq gas(s_1)) \vee (vc(s_0) \neq vc(s_1)) \vee (cc(s_0) \neq cc(s_1)) \vee (lc(s_0) \neq lc(s_1))\}$$

In addition, the program cannot reach certain states in which central/local exhaust is uncontrollable ($cc = 0$, $lc = 0$) while it is turned on ($c = 1$, $l = 1$). Thus

$$Sf_{bs} = \{s | (\neg cc(s) \wedge c(s)) \vee (\neg lc(s) \wedge l(s))\}$$

Notation. We use $\langle gas, vc, cc, lc | v, c, l \rangle$ to represent the state. If a variable value is '$*$', it means it can be either 0 or 1. Thus, $\langle 1, 1, 1, 1 | *, 0, 1 \rangle$ denotes a state where gas is present; vent cover, central exhaust and local exhaust are controllable; vent cover may be open or closed, central exhaust is not in use and local exhaust is in use.

Specification.

There are two major requirements considered critical in the system design. First of all, when hazardous gas is detected, the lab should be isolated from the building (denoted as R_{iso}) to prevent gas leak from the lab to the rest of the building. Second, the poisonous gas must be exhausted from

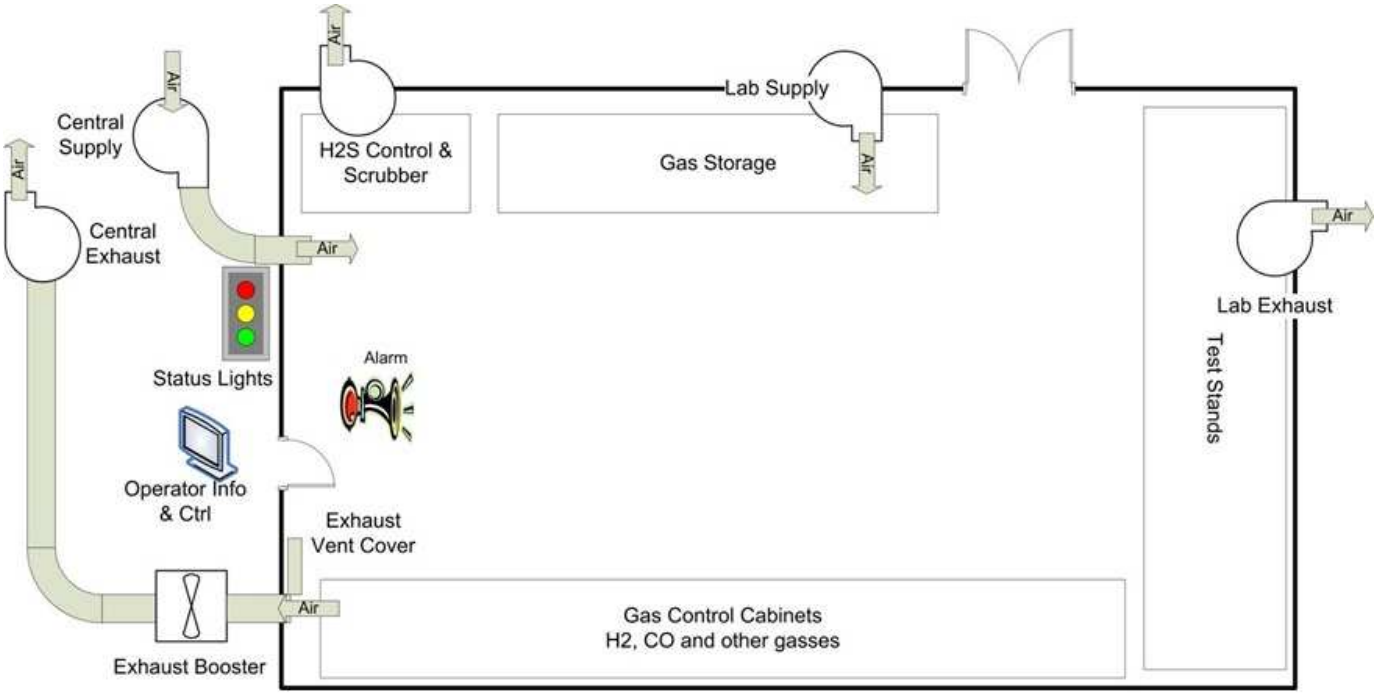

Fig. 8: Overview of OCRC Lab Safety System (from [15])

lab (denoted as R_{exh}) to outside air. These requirements are captured by R_{iso} and R_{exh}, where

- $R_{iso} = gas \Rightarrow \neg v$, gas isolated.
- $R_{exh} = gas \Rightarrow ((v \wedge c) \vee l)$, gas exhausted.

R_{exh} is considered much more important than R_{iso} since gases like hydrogen and carbon monoxide can explode. Hence, we can consider three requirements that are used in describing the graceful behavior.

- $Sf_1 = R_{iso} \wedge R_{exh}$, where Sf_1 is the ideal requirement that should be satisfied whenever possible. In this state, any leaked gas is exhausted while isolating the rest of building from gas. This corresponds to status *Safe* described at the beginning of this section.
- $Sf_2 = R_{exh}$, where Sf_2 is desirable when Sf_1 cannot be met, i.e., if isolation cannot occur then exhaust must be provided. This corresponds to the status *NotLocal* described at the beginning of this section.
- $Sf_3 = R_{iso} \vee R_{exh}$, where Sf_3 specifies that at least one of the two requirements should be met. This corresponds to the status *UnsafeLab* described at the beginning of this section.

Remark. For brevity of presentation, we do not model UnsafeCentral. Modeling UnsafeCentral will require three different levels of graceful degradation. However, our algorithm can be easily applied in deriving this third level of graceful degradation.

Original program.

In the ideal scenario, there is no gas leak and at least one of the exhaust systems is functioning correctly. Moreover, since no gas is leaked, the vent is always open. For efficiency reasons, whenever possible, the program chooses the central exhaust. But it switches to the local exhaust if needed. Thus, the program consists of three actions. Specifically, the first action turns on the central exhaust and turns off the local exhaust when both are available. If only one of the exhaust is available, the last two actions utilize the corresponding exhaust. The three actions of the program are as shown in Table I.

B. Application of Multi-Graceful Degradation in OCRC

In this section, we illustrate Algorithm 4 in the context of OCRC. First, we discuss the faults in different severity and the expected requirements in the presence of these faults. Then, specify corresponding relaxed specifications. In particular, we split the whole synthesis progress into four parts following Algorithm 4. In the end, we show the resulting program and compare it with the original OCRC system design.

1) Faults: Before adding graceful degradation to the program in Table I, we identify the faults that may require one to provide a weaker specification. We consider four possible faults that affect the system. Of these, the first fault, f_1, corresponds to the case where a gas leak occurs. The next two faults, f_2 and f_3, cause the central and the local exhaust to become uncontrollable. When the exhaust becomes unavailable/uncontrollable, the system receives a notification upon which the system must turn the corresponding exhaust to off. Moreover, it cannot turn that exhaust on again (except under manual intervention). Note that the assumption in this system is that both the local and the central exhaust do not become uncontrollable. This is captured in the modeling of fault actions f_2 and f_3. Finally, the last fault action, f_4, causes the vent cover to be uncontrollable. This is a stuck-at fault that causes the vent to remain in its current position. Its status

$\neg gas \wedge lc \wedge cc \wedge (\neg c \vee l) \longrightarrow c := 1, l := 0$ central exhaust is turned on if both controllable
$\neg gas \wedge \neg lc \wedge cc \wedge \neg c \longrightarrow c := 1$ central exhaust is turned on if local exhaust is uncontrollable
$\neg gas \wedge lc \wedge \neg cc \wedge \neg l \longrightarrow l := 1$ local exhaust is turned on if central exhaust is uncontrollable

TABLE I: Actions of Fault-intolerant Program p

cannot be changed by the program. These faults are as shown in Table II.

2) Requirements in the presence of Faults: In this system, the first graceful program, say p_{gft_1}, is designed for the case where faults f_1 and f_2 occur. Observe that in the presence of f_1 and f_2, local exhaust and vent cover are still controllable. Hence, we can eventually ensure both R_{iso} and R_{exh}. Note that in the presence of f_1 and f_2, system may reach states that violate Sf_1 if f_1 occurs while the central exhaust is being used. But all such states are still in Sf_2. Hence, the desired graceful behavior in this context is one that satisfies the safety specification Sf_2 and liveness specification $Lv_2 := true \rightsquigarrow Sf_1$.

The second graceful program, say p_{gft_2}, is designed for the case where all four faults can happen. If the local exhaust is uncontrollable then it would be impossible to satisfy Sf_1 when a gas leak occurs. Likewise, if the vent cover cannot be closed, it would be impossible to satisfy Sf_1. Hence, if the system is exposed to all four faults, the system may reach states that violate Sf_2. However, all such states still satisfy Sf_3. Hence, the desired graceful behavior in this context is one that satisfies the safety specification Sf_3 and liveness specification $Lv_2 := true \rightsquigarrow Sf_2$.

Thus, the system design requires us to consider two classes of faults. The first class includes actions f_1 and f_2. These faults are combined into one class since requirements in their presence are the same. And, the second class includes f_1, f_2, f_3 and f_4.

C. Application of Algorithm 4

First, we identify the inputs for Algorithm 4.

- The fault-intolerant program p is same as that shown in Table I.
- The invariant I is set of reachable states by p, which in particular are represented as $I = \{s | \neg gas(s) \wedge (l(s) \vee (c(s) \wedge v(s)))\}$
- In this case study, graceful behavior is desired only if gas is present. Thus the set of state predicates $S_{a1}, S_{a2}, \cdots, S_{an}$ are all set to $\{\langle 1, *, *, * | *, *, * \rangle\}$.
- Faults are as described in Section X-B1 and the relaxed specification is as described in Section X-B2.

Recall that Algorithm 4 utilizes a multi-step approach where in each step, one class of faults is considered. The Algorithm 4 invokes Algorithm 1 on Line 3. This subroutine expands the behavior of the original program to add new behaviors that only satisfy the weaker specification. Subsequently, it utilizes this program to add fault-tolerance in Line 4. Thus, for ease of understanding, we partition the application of Algorithm 4 into four parts, given below:

Part 1. (Line 3 in first iteration of loop 2-13): Generate graceful program p_{g_1}, such that it satisfies safety specification Sf_2 and liveness specification $Lv_2 := true \rightsquigarrow Sf_1$.

Part 2. (Line 4 in first iteration of loop 2-13): Add fault-tolerance to p_{g_1} to obtain p_{gft_1}.

Part 3. (Line 3 in second iteration of loop 2-13): Generate graceful program p_{g_2}, such that it satisfies safety specification Sf_3 and liveness specification $Lv_3 := true \rightsquigarrow Sf_2$.

Part 4. (Line 4 in second iteration of loop 2-13): Add fault-tolerance to p_{g_2} to obtain p_{gft_2}.

Part 1: Generation of Graceful Program p_{g_1}.

To generate p_{g_1} (on Line 3 of Algorithm 4), we invoke procedure $Gen\_graceful$. Based on the inputs to Algorithm 4, the inputs to the algorithm are as follows.

- S_a: As mentioned, $S_{a1} = \{\langle 1, *, *, * | *, *, * \rangle\}$.
- p: In the first iteration (2-13), the program transitions for $gen\_graceful$ is the fault-intolerant program p.
- I: Same as the invariant identified for fault-intolerant program.
- $spec_{r_1}$: The relaxed specification is a pair of safety specification and liveness specification.
  - Relaxed Safety: We specify safety specification as the set of bad states and bad transitions. First of all, we require that program always satisfies Sf_2, thus states in $\neg Sf_2$ are not allowed. In addition, the relaxed safety specification should always contain structural constraints Sf_{bs} and Sf_{bt}.
  - Relaxed Liveness: We require program ensure both exhaustion and isolation eventually, i.e. reach a state satisfying Sf_1.

  Hence $spec_{r_1} := \langle (\neg Sf_2 \cup Sf_{bs}, Sf_{bt}), true \rightsquigarrow Sf_1 \rangle$.

First on Line 1, we identify all the states in S_a satisfying $Sf_{r_{bs}}$. These states are denoted as $I_\Delta := ((v = 1 \wedge c = 1) \vee (l = 1)) \wedge (cc = 0 \rightarrow c = 0) \wedge (lc = 0 \rightarrow l = 0)$. Then, I' includes all states in I and I_Δ. On Line 3, we obtain the set of transitions satisfying the safety specification. In particular, we show all the transitions in Figure 9 (including solid and dashed arrow). Note that on Line 3, each state in I_Δ is added with a self-loop transition. Now we discuss the synthesis by loop (from Line 4 to Line 13) in different iterations.

- **Iteration #1:** By construction I only includes states where no hazardous gas is present and I_Δ only includes states where hazardous gas is present. The program

f_1: $\neg gas \longrightarrow gas := 1$	hazardous gas is detected
f_2: $cc \wedge lc \longrightarrow cc := 0, l := 1, c := 0$	central exhaust loses its control
f_3: $cc \wedge lc \longrightarrow lc := 0, l := 0, c := 1$	local exhaust is loses its control
f_4: $vc \longrightarrow vc := 0$	vent cover loses its control

TABLE II: Faults

cannot write the variable gas. Hence no transition from I_Δ to I is added on Line 7. There is no deadlock state identified in this iteration on Line 8. To realize the effect of Line 11, we first assign the rank (using function $rank$) over each state. According to Lv, states in Sf_1 are assigned rank 0. Since both states $\langle 1,1,1,1|1,1,1\rangle$ and $\langle 1,1,1,1|1,0,1\rangle$ reach some states in Sf_1 in one step, their ranks are 1. Therefore, the transition between them is removed on Line 11. Likewise, on Line 11, the self-loop transitions on states $\langle 1,1,1,1|1,1,0\rangle$, $\langle 1,1,1,1|1,1,1\rangle$, $\langle 1,1,1,1|1,0,1\rangle$ and $\langle 1,1,0,1|1,0,1\rangle$ are removed. On Line 12, those states with ranks of ∞ are removed, i.e. those underlined states in Figure 9.

– **Iteration #2:** Algorithm 1 applies no change in the next iteration, thus the loop terminates.

Therefore the remaining transitions (and states) together with those in p (and in I) are returned as p_{g_1} (and I_{g_1}). We show transitions of p_{g_1} (except those shown in Table I) in Figure 9.

Part 2: Adding Fault-tolerance to p_{g_1}.

To add fault-tolerance, we use procedure $Add\_masking$ as a black box. Thus to illustrate the result of adding fault-tolerance, we first consider how the program can be perturbed by faults, that is set of reachable states by p_{g_1} in the presence of f, where $f = f_1 \cup f_2$; then show that starting from all these reachable states there are transitions provided by either original program p or graceful program p_{g_1}.

In the absence of faults, if only f_1 occurs, the system reaches one of the states in $\langle 1,1,1,1|*,*,*\rangle$. Note that all these states are generated in p_{g_1} and have outgoing transition, thus f_1 is tolerated. In the absence of faults, if only f_2 occurs, following transitions in p, system switches to local exhaust, thus f_2 is tolerated. If original program perturbed by f_1 and f_2 together, the only state can be perturbed to is $\langle 1,1,0,1|1,0,1\rangle$ which is generated in p_{g_1} and has an outgoing transition. Thus, there is no changes on Line 4 in Algorithm 4.

Part 3: Generation of Graceful Program p_{g_2}.

In this case, we add fault-tolerance to the program generated in Part 2. The inputs of program and invariant are selected from the first iteration. Faults consist of all four faults in Table II and the specification in the presence of faults is as specified in Section X-B2.

We do not elaborate the effect of Algorithm 1 again on p_{g_2}. Instead, we show set of newly discovered transitions and states in p_{g_2} (not including those in p_{g_1}) in Table III.

Part 4: Adding Fault-tolerance to p_{g_2}. There are no new transitions added by $Add\_masking$, and p_{gft_2} is same as p_{g_2}.

D. Extension to deal with UnsafeCentral

To finalize the synthesis, we also consider the status of *UnsafeCentral*. In this case, neither isolation nor exhaust may be satisfied, at least temporarily. Hence, the relaxed requirement does not include any safety specification. However, it includes a liveness specification, namely, $true \rightsquigarrow Sf_3$. Although we do not present the application of Algorithm 4, we can easily obtain the corresponding program by considering only another class of faults and applying Algorithm 4 in a third iteration.

The state where Sf_3 is violated is reached due to faults such as transients. It could also be reached if both exhaust systems fail temporarily. Algorithm 4 already handles this scenario as well. Specifically, for this case, we need to model fault f_5 as a transient fault that requires the liveness specification $true \rightsquigarrow Sf_3$. If we add this as the third fault class and apply Algorithm 4, the resulting program is same as the *ventilation* system (i.e., the system except alarms, status lights etc) of OCRC.

XI. Related Work

The work in this paper is closely related to that of controller synthesis, game theory and automated addition of fault-tolerance. Controller synthesis considers the following problem: Given two languages $\mathcal{U}$ (plant) and $\mathcal{D}$ (desired system), identify a third language $\mathcal{C}$ (controller), such that $\mathcal{U} \cap \mathcal{D} \subseteq \mathcal{C}$ [16]. Thus, the goal is to begin with the *plant* and add *controller* to obtain the desired system. The idea of transforming a fault-intolerant system into a fault-tolerant system using controller synthesis is used in [17]. Also in [18], Girault and Rutten demonstrate the application of discrete controller synthesis in automated addition of fault-tolerance in the context of untimed systems. Our work in this paper differs from this work in that we are trying to relax the specification of the given system whereas they are trying to strengthen it. In the context of game theoretical approach for model revision, a program is automatically fixed as a game [19], [20]. The game is played on the model of two players [21], i.e., program and environment. A program is considered to win the game if the specification is always satisfied no matter how the environment interacts with the program. Game theoretic methods are usually based on the theory of tree automata [22].

Model repair for probabilistic system [3] is to revise a probabilistic system M such that the new system M' satisfies a probabilistic temporal logic formula. M' differs from M only in the transition flows of controllable states. Our work is orthogonal to that in [3] in that this work can be extended in the context of dealing with probabilities and their work can be extended to deal with addition of graceful degradation. Algorithms for automatic addition of fault-tolerance [1], [5] add fault-tolerance concerns to existing untimed or real-time programs in the presence of faults, and guarantee the addition of no new behaviors to the original program in the absence

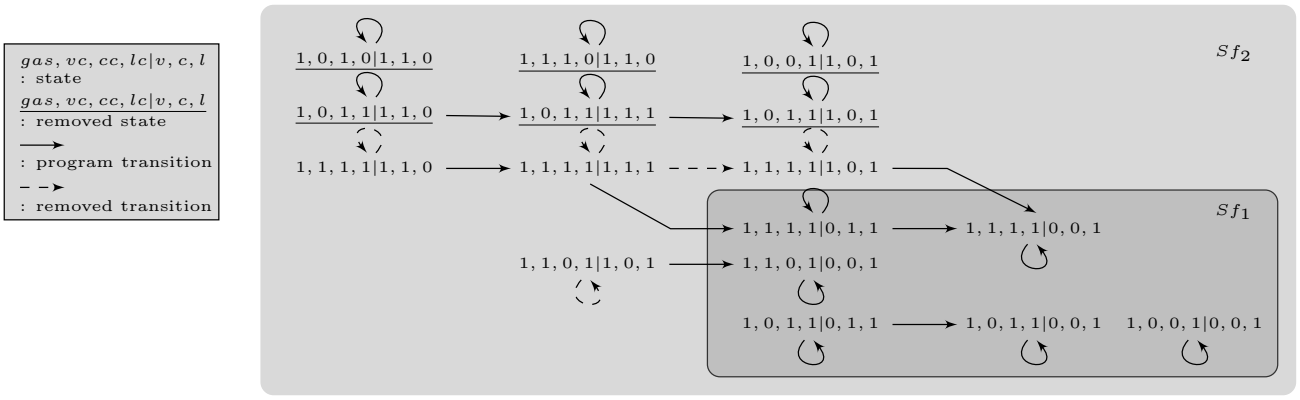

Fig. 9: Transitions of p_{g_1} (except those shown in Table I)

$\langle 1,0,1,1\|0,1,0\rangle \rightarrow \langle 1,0,1,1\|0,1,1\rangle$	$\langle 1,0,0,1\|0,0,0\rangle \rightarrow \langle 1,0,0,1\|0,0,1\rangle$
$\langle 1,0,1,1\|0,0,0\rangle \rightarrow \langle 1,0,1,1\|0,0,1\rangle$	$\langle 1,1,1,0\|0,1,0\rangle \rightarrow \langle 1,1,1,0\|1,1,0\rangle$
$\langle 1,1,0,1\|0,0,1\rangle \rightarrow \langle 1,1,0,1\|0,0,1\rangle$	$\langle 1,1,1,1\|0,0,0\rangle \rightarrow \langle 1,1,1,1\|0,0,1\rangle$
$\langle 1,1,1,1\|0,1,0\rangle \rightarrow \langle 1,1,1,1\|0,1,1\rangle$	$\langle 1,1,1,0\|0,0,0\rangle \rightarrow \langle 1,1,1,0\|0,1,0\rangle$

TABLE III: Newly discovered transitions in **Part 3**

of faults. In the context of this paper, we utilize the synthesis algorithm for adding fault-tolerance. Our work builds on this work by enabling repair in scenarios where the previous work fails to perform repair or has a much higher overhead.

XII. Discussion and Summary

A. Discussion

Order of Operations in Different Phases. In our approach, we first designed a program that satisfied degraded specification. Subsequently, we added fault-tolerance to the resulting program. A natural question, thus, is whether it would be possible to add fault-tolerance as the first step. We argue that this is not possible. In particular, the main reason for using the approach in this paper is for cases where it is impossible for the fault-tolerant program to recover to states from where the *original* specification is satisfied.

Comparison with Algorithms for Adding Safety, Liveness or Fault-tolerance. Existing work [5], [13], [23] focuses on adding properties to an existing program. Since the goal of this work is to construct a program that preserves the existing specification (in the absence of faults) *and* to satisfy the newly desired property, they prohibit transitions to be added in the absence of faults. This is due to the fact that adding transitions creates new program behaviors that may not satisfy the original (universal) specification. However, these approaches permit removal of transitions, as removal of transitions preserves existing universal specification as long as it does not create deadlocks. Our solution in Algorithm 1 is explicitly designed to add such transitions so that the new program satisfies a relaxed specification. For this reason, the algorithms in this paper are more general than that in [5], [13], [23]. Specifically, the algorithm in this paper is applicable even if it is impossible to recover the system to states from where the original specification is satisfied. By contrast, the algorithms in [5], [13], [23] will declare failure in these cases.

Scalability of Approach. State space explosion is generally unavoidable in the area of program verification or synthesis. Even though the addition of fault tolerance (FT) is NP-complete in the size of the state space, it has been shown to be feasible to add FT to programs of moderate size [24], up to programs with state space of 10^{100} [25]. However, the approach may not directly scale to C++ programs of 1000s of lines of code. One possible way to circumvent this is through model extraction. In [26], we have extracted the necessary state transition model from a UML model, apply fault tolerance transformation to it and convert it back (with some heuristics) to a UML model. In [27], the authors have shown how an UPPAAL model can be extracted from a SystemC model (an extension of C). We have shown how FT can be added to these models [28] and how it can be sliced to find subset of the model relevant to the property at hand. In this context, automated reverse transformation is not yet available.

Transformation Failure. Our algorithm involves 2 steps: (i) Addition of graceful behavior and (ii) addition of fault-tolerance. Technically, the first step cannot fail. It can simply return the original program, i.e., it may not add any new behaviors. This may, however, cause the second step to fail. Adding fault-tolerance fails for two reasons: (1) it is impossible to satisfy the property or (2) heuristics used for adding fault-tolerance fails. For high atomicity programs –where a process can read all the variables–, the algorithms for adding fault-tolerance are sound and complete, i.e., they declare failure only if a solution is not feasible. For distributed programs, the problem of adding fault-tolerance is NP-complete. Hence, heuristic based approaches have been developed for adding fault-tolerance in distributed systems.

Implication to Concrete Programs. Our work focuses on event based programs where each process consists of several event processing modules that responds to events (that may correspond to external events or internal events indicating that certain predicate is true). In this case, adding behaviors

corresponds to adding new modules for dealing with newer predicates. By contrast, removing behaviors corresponds to restricting ore removing existing event response modules.

An alternate approach is to utilize ideas such as recovery blocks [29]. Specifically, in this approach, at certain instances of the program, acceptance conditions are checked. If they are violated, alternate code is executed to recover from the resulting faults/errors. New behaviors can be added by adding such recovery blocks. Also, in [30], authors present an approach to utilize concurrent atomic actions (that are the basis of our model) instead of recovery blocks.

Interference Among Properties. In our case study on multi-graceful degradation, we assumed that the specifications were in a strict order, with the original specification being strongest, the next one weaker and so on. This allowed us to handle the stepwise addition where we consider one specification at a time. It is possible to consider a version where you have incompatible weaker specifications. For example, one specification could be to satisfy requirement 1 alone but not necessarily requirement 2. Another specification could be to satisfy requirement 2 but not necessarily requirement 1. We believe that designing graceful degradation in this manner is expected to be more difficult. We anticipate it being NP-complete based on a result in [31]. In this work, authors have considered adding failsafe and nonmasking fault-tolerance. They are similar to incompatible requirements. We anticipate that such scenarios are generally rare; in most cases, requirements can be characterized in terms of their priority and the desired graceful behaviors require that highest priority requirements should be satisfied. Hence, the graceful behaviors naturally fall into (1) drop 1 requirement with lowest priority, (2) drop 2 requirements with lowest priority and so on.

B. Conclusion

Existing approaches for automated addition of fault-tolerance have focused on the following problem: *Given a fault-intolerant program p, construct a fault-tolerant program p' such that p' behaves like p after recovery from faults*. It follows that after recovery is complete, p' satisfies the original specification of p. However, as discussed in the paper, there are several instances where it is impossible (or undesirable) to recover the system to states from where the original specification is satisfied. Hence, we focused on the problem of graceful fault-tolerance, where the goal is to construct a program p' such that (1) in the absence of faults p' behaves like p, (2) p' provides desired fault-tolerance, and (3) after recovery is complete, p' exhibits graceful behavior where it satisfies the given weaker specification.

We presented a two-phase approach for designing graceful fault-tolerance: In the first phase, we transformed the input fault-intolerant program p into a graceful program p_g that includes all behaviors of p as well as new behaviors that satisfy the desired weaker specification. In the second phase, we used both p and p_g to obtain a fault-tolerant program p' that behaves like p in the absence of faults and recovers to behaviors provided by p_g after the faults stop.

To solve this problem, we have first formalized the problem of adding graceful degradation to a program and have then provided an algorithm that transforms a given fault-intolerant program into a gracefully degrading program. We presented two solutions, one for concurrent programs and another for distributed programs where each process can read or write only a subset of program variables. For the second phase, we have formulated the problem of addition of fault-tolerance where one begins with two (or possibly more) programs that satisfy successively weaker specifications.

We have then demonstrated the applicability of our algorithms through three case studies: (i) a printer system [4], (ii) cellular networks [11] and (iii) Byzantine agreement. Of these, if we used the first two case studies with existing approaches [23] then they will declare failure since there does not exist a fault-tolerant program that satisfies the original specification. And, for the third case study, a significantly more complex input is required to permit addition of fault-tolerance.

We have then extended the notion of graceful degradation to that of multi-graceful degradation whereby a program can satisfy some weaker specification based on the severity of the fault type being present. Similarly, we have first formalised the problem and provided a polynomial-time algorithm that automates the addition of multi-graceful degradation. We have shown the applicability of the algorithm on a non-trivial case study, namely the OCRC system.

Our approach for designing graceful programs also opens up new research directions in the area of program repair, where the goal is to add a desired property (safety, liveness, fault-tolerance, etc.) to a given program. However, these approaches fail if the newly desired property is incompatible (e.g., adding priorities to programs that assume equal fairness) with an existing property of the program.

For example, consider the case where we have a program that provides fair mutual exclusion access to processes $P1$ and $P2$. Suppose we want to add the property that requires that $P1$ has higher priority than $P2$. Since the new requirement is incompatible with existing program properties, existing approaches that add a new property while preserving existing universal properties fails. In these cases, our first phase can be used to remove 'fairness' from the given program so that the priority requirement can be added. Thus, novel algorithms to add new properties to an existing program after removing the offending property are required.

REFERENCES

[1] B. Bonakdarpour and S. S. Kulkarni, "Exploiting symbolic techniques in automated synthesis of distributed programs," in *In IEEE International Conference on Distributed Computing Systems*, 2007, pp. 3–10.

[2] F. Abujarad and S. Kulkarni, "Constraint based automated synthesis of nonmasking and stabilizing fault-tolerance," in *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*, sept. 2009, pp. 119 –128.

[3] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka, "Model repair for probabilistic systems," in *TACAS*, 2011, pp. 326–340.

[4] M. Herlihy and J. M. Wing, "Specifying graceful degradation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 1, pp. 93–104, 1991.

[5] S. S. Kulkarni and A. Arora, "Automating the addition of fault-tolerance," in *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, 2000, pp. 82–93.

[6] W. Leal, M. McCreery, and D. Faria, "The ocrc fuel cell lab safety system: A self-stabilizing safety-critical system," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in

Computer Science, X. Défago, F. Petit, and V. Villain, Eds. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 326–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24550-3_25

[7] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181 – 185, 1985.

[8] E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communications of ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[9] A. Tahat and A. Ebnenasir, "A hybrid method for the verification and synthesis of parameterized self-stabilizing protocols," in *Proceedings LOPSTR*, 2014, pp. 201–218.

[10] A. Klinkhamer and A. Ebnenasir, "Verifying livelock freedom on parameterized rings and chains." in *Proceedings Stabilization, Safety and Security of Distributed Systems*, 2013, pp. 163–177.

[11] G. v. Zruba, I. Chlamtac, and S. K. Das, *A Prioritized Real-TimeWireless Call Degradation Framework for Optimal Call Mix Selection*. Kluwer Academic publishers, 2002.

[12] S. S. Kulkarni and A. Ebnenasir, "Complexity issues in automated synthesis of failsafe fault-tolerance," *IEEE Trans. Dependable Sec. Computing*, vol. 2, no. 3, pp. 201–215, 2005.

[13] F. C. Gärtner and A. Jhumka, "Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications," in *FORMATS/FTRTFT*, ser. Lecture Notes in Computer Science, Y. Lakhnech and S. Yovine, Eds., vol. 3253. Springer, 2004, pp. 183–198.

[14] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[15] W. Leal, M. McCreery, and D. Faria, "The ocrc fuel cell lab safety system: a self-stabilizing safety-critical system," in *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 326–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050638

[16] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[17] K. H. Cho and J. T. Lim, "Synthesis of fault-tolerant supervisor for automated manufacturing systems: A case study on photolithography process," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 2, pp. 348–351, 1998.

[18] A. Girault and É. Rutten, "Automating the addition of fault tolerance with discrete controller synthesis," *Formal Methods in System Design (FMSD)*, vol. 35, no. 2, pp. 190–225, 2009.

[19] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Principles of Programming Languages (POPL)*, 1989, pp. 179–190.

[20] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Conference on Computer Aided Verification (CAV)*, 2005, pp. 226–238, LNCS 3576.

[21] W. Thomas, "On the synthesis of strategies in infinite games." in *Theoretical Aspects of Computer Science (STACS)*, 1995, pp. 1–13.

[22] ——, *Handbook of Theoretical Computer Science: Chapter 4, Automata on Infinite Objects*. Elsevier Science Publishers B. V., 1990.

[23] F. A. Borzoo Bonakdarpour, Sandeep S. Kulkarni, "Symbolic synthesis of masking fault-tolerant distributed programs," *Distributed Computing*, vol. 25, no. 1, pp. 83–108, 2012.

[24] F. Faghih and B. Bonakdarpour, "Smt-based synthesis of distributed self-stabilizing systems," *Trans. Adaptive and Autonomic Systems (TAAS)*, vol. 10, no. 3, pp. 1–26, 2015.

[25] B. Bonakdarpour, S. Kulkarni, and F. Abujarad, "Symbolic synthesis of masking fault-tolerant distributed programs," *Distributed Computing*, vol. 25, no. 1, pp. 83–108, 2012.

[26] J. Chen and S. S. Kulkarni, "Mr4um: A framework for adding fault tolerance to uml state diagrams." *Theoretical Computer Science*, vol. 496, pp. 17–33, 2013.

[27] R. Hajisheykhi, A. Ebnenasir, and S. S. Kulkarni, "Evaluating the effect of faults in systemc tlm models using uppaal," in *Proceedings SEFM*, 2014, pp. 175–189.

[28] ——, "Ufit: A tool for modeling faults in uppaal timed automata," in *Proceedings NFM*, 2015, pp. 429–435.

[29] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 221–232, 1975. [Online]. Available: https://doi.org/10.1109/TSE.1975.6312842

[30] B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, Z. Wu, and J. Xu, *From Recovery Blocks to Concurrent Atomic Actions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 87–101. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-79789-7_6

[31] A. Ebnenasir and S. Kulkarni, "Feasibility of stepwise design of multitolerant programs," *ACM Trans. Softw. Eng. Methodology*, vol. 21, no. 1, pp. 1–49, 2011.