Automatic Generation of Graceful Programs

Yiyan Lin
Department of Computer Science and Engineering
Michigan State University
East Lansing, USA
linyiyan@cse.msu.edu

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

***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 requirements.**

In this paper, we focus on automatically revising a given program to obtain a corresponding graceful program, i.e., a program that satisfies a weaker specification. Specifically, this step involves adding new behaviors that satisfy the given subset of specification. Moreover, it ensures that during this process, it does not remove any behavior from the original program. With this motivation, in this paper, we focus on automatic derivation of the graceful program, i.e., a program that contains all behaviors of the original program and some new behaviors that satisfy the weaker conditions. We note that this aspect differentiates this work from previous work on controller synthesis as well as automated addition of fault-tolerance in that this work requires that no new behaviors are added in the absence of faults.

I. Introduction

We are increasingly dependent on highly available computer systems to provide fully functional and stable services. However these existing programs are often subjected to new types of faults that are not considered the original design. This may occur 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 fault-tolerance and that the existing functionality of the program is preserved.

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. In all these levels, if the fault perturbs the program then it is guaranteed to recover to legitimate states from where it satisfies its specification. Masking fault-tolerance includes an additional requirement that the safety property is preserved during such recovery process. And, stabilizing fault-tolerance requires that even if faults perturb the program to an arbitrary state, the program still recovers to legitimate 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. In such scenarios, it is desirable that the program exhibits some graceful behavior, i.e., behavior that is close to its original behavior. A canonical example of a graceful behavior program includes the case where the original program provides core services and auxiliary services. However, after recovery, the program may only provide core services and fail to provide the auxiliary services. More generally, in such a system, it is expected that the program will satisfy its specification under normal circumstances. However, upon occurrence of faults, it may satisfy (a given) weaker (respectively, relaxed) specification.

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 2^n possible specifications for each possible subset of these n properties. Thus, theoretically, we can consider 2^n possible programs depending upon the exact subset of properties that are expected to be satisfied under different circumstances. In practice, however, considering such 2^n specifications is unnecessary. For example, in the above scenario, there is no need to consider a program that only provides auxiliary services. Likewise, there is also no need to consider a program that provides neither the core nor the auxiliary services. In other words, in the above example, we need to consider two programs, one that guarantees that both the core and auxiliary services are provided, and another that guarantees that at least the core services are provided.

To further illustrate application of graceful degradation, we use the printer example shown 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, one may prefer

This work is supported by NSF CNS-0914913 and AFOSR Award FA9550-10-1-0178.

that printer system ensure that the printing occurs in FIFO order, i.e., the task that is dequeued first is printed first. Thus, in an ideal setting, the specification requires that at most one dequeue operation can occur at a time. And, the next dequeue operation occurs only after the current task is printed. As one can imagine, although this is a very simple specification, it results in reduced availability. Moreover, in the presence of faults such as network delays, 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, the simplest such specification is that task n is printed only after task $n-2$ is printed although task n may be printed before task $n-1$.

Based on the above examples, we can view a graceful degradation program to have the following properties. In the absence of faults, the program satisfies its original (stronger) specification. However, if faults occur then it is guaranteed to recover to states from where it satisfies the degraded (weaker) specification. Our goal in this paper is to focus on automated addition of graceful degradation to an existing program. Thus, in our algorithm, we begin with the fault-intolerant program that satisfies the original specification, the original and degraded specification and faults. And, the goal of the algorithm is to obtain a fault-tolerant graceful degradation.

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. Asking the designer to specify such a program is undesirable, since it increases the overhead for the designer.

With this motivation, in this work, we focus on generation of a graceful program where we begin with a program, say p, that satisfies the original specification and constructs a program, say p_g, that satisfies the weaker specification. Clearly, p_g cannot satisfy the weaker specification in an arbitrary way. Hence, we constrain this step to require that p_g has all transitions of program p. However, it may include additional transitions that allow it to exhibit behaviors that only satisfy the weaker specification.

Revisiting our previous example to illustrate generation of graceful programs, we begin with a program that ensures that the print ordering is FIFO. Our work will generate a program that will include the FIFO behavior as well as the behavior where some limited out-of-order printing is permitted. In other words, the generated program will not prevent FIFO behavior but it will not guarantee it. Additionally, it will prevent excessive out-of-order printing.

Observe that the generation of the graceful program does not take into consideration the effect of fault that may require one to provide graceful degradation. Instead, it generates a program, p_g, whose behaviors could be used to provide recovery in the presence of faults. Although this part is outside the scope of this paper due to reasons of space, one can design a fault-tolerant graceful degradation program p_f using the algorithm in this paper. Specifically, in this case, we use the input of the original program, p, the graceful program, p_g, and faults as input to generate p_f such that p_f behaves like p in the absence of faults. However, in the presence of faults, it recovers to the behaviors of p_g (instead of p). We refer the reader to [5] for this extension.

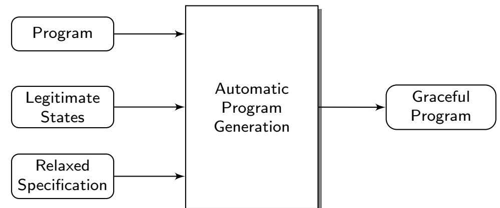

Figure 1: Design of Graceful Program

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

- We define the problem of automated addition of graceful degradation and present an algorithm to solve it. Unlike previous algorithms [6] 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 illustrate our algorithm with one case study, namely the printer system from [4].

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 one case study to demonstrate the graceful program generation step by step. Finally, in Section VI we discuss related work and in Section VII, we present the conclusion and discuss future 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 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. In the remaining part of this paper, 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. Specification

Following Alpern and Schneider [7], we let the specification of program consisting of a safety specification and a liveness specification. The safety specification is specified in terms of a set of bad states, say Sf_{bs}, that program is not allowed to reach, and a set of bad transitions, Sf_{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) $\forall j : 0 \leq j < length(\sigma) : s_j \notin Sf_{bs}$, and (2) $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j) \notin Sf_{bt}$.

The liveness specification, on the other hand, denotes "good thing" happens during program execution. We use *leadsto* property ($\mathcal{L} \leadsto \mathcal{T}$) 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 safety 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 $\langle Sf, Lv\rangle$, where Sf is a safety specification and Lv is a liveness specification. A sequence σ *satisfies* $spec$ iff it satisfies Sf and Lv. 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 invariant of p iff (1) S is closed on p; (2) Every computation of p that starts in a state, say s, where $s \in S$ satisfies $spec$; and (3) $S \neq \emptyset$.

C. 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 $spec_r$ then σ satisfies $spec$.

III. Problem Statement

In this section, we formally define the problem of generation of program that satisfies the weaker specification (cf. Problem 1). We begin with a program p that satisfies the specification $spec$ from I. We derive a 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 [6] in that this enlarges the invariant by adding new states from where the new specification can be satisfied. By contrast, in [6], 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

Specifically, 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.

Our program 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.

The algorithm first computes $I_\triangle$ to be the set of states in S_a except those that violate safety (i.e., those in $Sf_{r_{bs}}$). The first guess for I_g, the invariant for the graceful program is then set to $I \cup I_\triangle$ (Line 2). Then, the algorithm 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 $I_\triangle$ unless they violate safety $Sf_{r_{bt}}$ (Line 3). Starting from Line 4, the algorithm revises the program by

ensuring the liveness specification is satisfied. From Line 6 to 9, for the generated program we first exclude the deadlock states, i.e., states from where no transitions originate 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 Line 11 and 12, we exclude transitions where rank does not decrease. Removal of such transitions ensures that there will be no cycles that prevent the program from reaching $\mathcal{T}$. One potential side effect of removing transitions is that the new program may include deadlock states. 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 specification $spec_r$ including weaker safety specification Sf_r (consisting of $Sf_{r_{bs}}$ and $Sf_{r_{bt}}$), and liveness specification Lv_r (consisting n leads-to properties of the form $\mathcal{F}_i \leadsto \mathcal{T}_i$, $i \in 1 \cdots n$).

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: $p' := \{(s_0, s_1)|s_0, s_1 \in I' :: (s_0 \in I \wedge (s_0, s_1) \in p) \vee (s_0 \in I_\Delta \wedge (s_0, s_1) \notin Sf_{r_{bt}})\}$
4: **repeat**
5: $\quad I_{old} := I'$
6: $\quad$**repeat**
7: $\quad\quad p' := maxp(p', I, I')$
8: $\quad\quad I' := I' - deadlock(I', p')$
9: $\quad$**until** $I_{old} = I'$
10: $\quad I_{old} := I', p_{old} := p'$
11: $\quad p' := p' - \bigcup_{i \in 1 \cdots n}\{(s_0, s_1)|s_0 \in I' \wedge rank(s_0, \mathcal{T}_i, p') \leq rank(s_1, \mathcal{T}_i, p') \wedge rank(s_0, \mathcal{T}_i, p') \neq 0 \wedge rank(s_1, \mathcal{T}_i, p') \neq \infty\}$
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.

Theorem 1. *Algorithm 1 is sound.*

V. Case Studies

In this section, we demonstrate process of generating the graceful program with one case study, namely the printer system. In the 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 step by step.

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.

To model this state space and transitions concisely, we use variables (associated with a domain) and actions. And, 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 involving program variables and $statement$ updates program variables. 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.

Original program. The program in [4] consists of several clients and several print servers. Using the transaction semantics, the clients 'enqueue' their print request in a shared queue. The print servers remove from this queue and print the task. For simplicity, we do not model the transactions used for concurrency control. Furthermore we omit the enqueue operation since it does not affect the behavior we interested in.

Hence, the original program, say A, 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 ith task has been printed ($p_i = 1$), then the next task ($i+1$) in queue get a chance to be dequeued ($d_{i+1} := 1$). Thus actions are as follows:

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

Safety Specification. Recall that the safety specification in terms of a set of bad states 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 generated 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,

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

Thus, we identify 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,

$Lv := true \leadsto S_L$, 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 A. For simplicity, we assume that there are three tasks in the system. Hence, the state of the program is represented as $(d_1 d_2 d_3, p_1 p_2 p_3)$, 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 A by starting from the initial state $(000, 000)$.
- State predicate S_a is instantiated to be $S_p - I$

First consider the candidate states generated outside I from Line 1 that also excludes those violate Sf_{bs_2}. In general, as the weaker safety specification requires, new program allows two tasks can be dequeued regardless either one finished printing or not and a new task can then be dequeued if either one of two previous dequeued tasks finishes printing. In other words,

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

The new program transitions generated are all possible pair of states in $I_\triangle$ plus those recovered from $I_\triangle$ 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 $I_\triangle$ 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 the context shown in figure 2. 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 2 shows the behavior of the graceful program. Note that in this program, the tasks being printed is not totally out of order, though it is not FIFO. For example, task 3 cannot precedes task 2 provided that task 1 has not been printed.

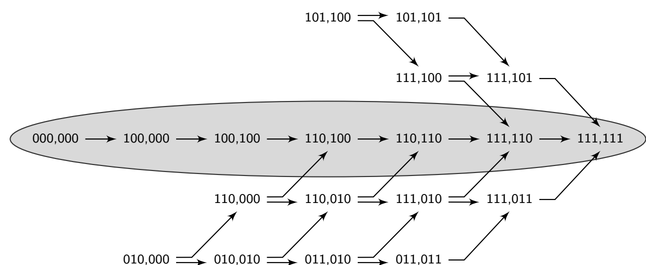

Figure 2: Printer System with original invariant $I = \{(000,000),(100,000),(100,100),(110,100),(110,110),(111,110),(111,111)\}$ and invariant (under weaker specification) $I_g = I \cup \{(110,000),(110,010),(111,010),(111,011),(110,110),(111,100),(111,101),(010,000),(010,010),(011,010),(011,011)\}$

VI. 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 $\mathcal{U} \cap \mathcal{D} \subseteq \mathcal{C}$ [8]. 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 was used in in [9]. Also in [10], 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, program is automatically fixed as a game [11], [12]. The game is played on the model of two players [13], 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 generally based on the theory of tree automata [14].

The 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], [6] 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 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 significantly higher overhead for the designer.

VII. Conclusion

In this paper, we presented the approach for generating the graceful program. We presented a sound algorithm for it. Additionally, we illustrated our approach with one example: *printer system*. The example is chosen from [4] where a similar graceful requirement is considered in the context of manual design.

As we saw from the case study, the algorithm tries to construct the graceful program to have maximal choice (non-determinism). This algorithm also differs from previous work on automated addition of fault-tolerance in that previous work on automated addition of fault-tolerance disallows *adding* transitions in the absence of faults. By contrast, in the design of graceful program, *removing* transitions is disallowed. This ensures that the graceful program has all the behaviors of the original program. Additionally, it adds new behaviors that satisfy the weaker specification. These new behaviors are crucial in providing recovery in the presence of faults.

Although it is outside the scope of this paper, it is possible to combine algorithms for adding fault-tolerance/multitolerance to the problem of designing graceful programs. Specifically, it is possible to add fault-tolerance to the printer system discussed in this paper. Other examples where this approach has been used include byzantine agreement and ventilation control at Ohio Coal Research Center (OCRC).

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] Y. Lin, S. Kulkarni, and W. Leal, “Tech-report: Automated multi-graceful degradation: A case study,” in *www.cse.msu.edu/~sandeep/publications/ocrctech/tech.pdf*.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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