On the Complexity of Synthesizing Relaxed and Graceful Bounded-Time 2-Phase Recovery [*]

Borzoo Bonakdarpour[1] and Sandeep S. Kulkarni[2]

[1] VERIMAG
Centre Équation
2 ave de Vinage
38610, Gières, France
Email: borzoo@imag.fr

[2] Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: sandeep@cse.msu.edu

Abstract. The problem of enforcing bounded-time 2-phase recovery in real-time programs is often necessitated by conflict between fault-tolerance requirements and timing constraints. In this paper, we address the problem of synthesizing two types of 2-phase recovery: relaxed and graceful. Intuitively, relaxed 2-phase recovery requires that in the presence of faults, the program recovers to an *acceptable* behavior within some time θ and recovers to *ideal* behavior within time δ. And, graceful 2-phase recovery allows us to capture a requirement that the time to recover from faults is proportional to the perturbation caused by that fault. We show that the problem of synthesizing relaxed bounded-time 2-phase recovery is NP-complete although a similar problem of graceful 2-phase recovery can be solved in polynomial-time both in the size of the input program's region graph. Finally, based on the results in this paper, we argue that the requirement of intermediate *recording* of a fault before reaching legitimate states can increase the complexity of adding fault-tolerance substantially.

Keywords: Fault-tolerance, Real-time, Bounded-time recovery, Phased recovery, Program synthesis, Program transformation.

1 Introduction

Achieving correctness is perhaps the most important aspect of using formal methods in design and development of computing systems. Such correctness

[*] This is an extended version of a paper appeared in the proceedings of FM'09: International Symposium on Formal Methods, LNCS, Springer-Verlag, 2009. This work is partially sponsored by the COMBEST European project, NSF CAREER CCR-0092724, and ONR Grant N00014-01-1-0744.

turns out to be a fundamental element in gaining assurance about reliability and robustness of safety/mission critical embedded systems. These systems are often real-time due to their controlling duties and integrated with physical processes in hostile environments. Hence, *time-predictability* and *fault-tolerance* are two desirable features of programs that operate in such systems. However, these features have conflicting natures, making achieving and reasoning about their correctness fairly complex.

One way to deal with this complexity is to design automated synthesis and revision algorithms that build programs that are correct-by-construction. Algorithmic synthesis of programs in the presence of an adversary has mostly been addressed in the context of timed controller synthesis (e.g., [12, 14, 4, 5]) and game theory (e.g., [13, 15]). In controller synthesis (respectively, game theory), program and *fault* transitions can be modeled as controllable and uncontrollable actions (respectively, in terms of two players). In both approaches, the objective is to restrict the actions of a *plant* or *player* at each state through synthesizing a *controller* or a *wining strategy*, such that the behavior of the entire system always meets some safety and/or reachability conditions. However, the notion of dependability and in particular fault-tolerance requires other functionalities that are not typically addressed in controller synthesis and game theory. For instance, fault-tolerance is concerned with *bounded-time recovery*, where a program returns to its normal behavior when its state is perturbed due to the occurrence of faults. In this context, a recovery mechanism is normally *added* to a program so that it reacts to faults properly.

Although synthesis algorithms are known to be intractable, it has been shown that their complexity can be overcome through:

- focusing on safety and liveness properties typically used in specifying systems rather than considering any arbitrary specification,
- rigorous complexity analysis for each class of properties to identify bottlenecks,
- devising intelligent heuristics that address complexity bottlenecks, and
- exploiting efficient implementation techniques.

By applying these principles, we have been able to synthesize distributed fault-tolerant programs with reachable states of size 10^{60} and beyond [9, 11], even though the worst case complexity (NP-completeness in the state space) initially seemed to be unfeasible to cope with in practice. In case of real-time dependable systems, however, the problem is more complex, because of conflicting nature of requirements and high complexity of decision procedures simultaneously.

In this paper, we focus on one aspect of the conflict between real-time constraints and fault-tolerance requirements. In particular, the fault-tolerance requirement of the program may require that eventually the program recovers to its legitimate states from where its subsequent behavior is *ideal*, i.e., one that could occur in the absence of faults. Also, the real-time constraints may require that the recovery to the ideal behavior be achieved quickly. When satisfying both these requirements is not feasible, one approach is to ensure that the program recovers quickly to an *acceptable* behavior and eventually recovers to its *ideal* behavior. The recovery requirements for *acceptable* and *ideal* behavior can be specified in terms of a set of states Q and S respectively. Thus, the requirements for a real-time fault-tolerant system can be viewed as a *2-phase recovery*, where the program eventually reaches Q within some time θ and eventually reaches S in some time δ.

There are different variations for such 2-phase recovery problem. The scenario discussed above can be expressed in terms of constraints of the form $(\neg S \mapsto_{\leq \theta} Q)$ and $(Q \mapsto_{\leq \delta} S)$, i.e., starting from any state in $\neg S$, the program first recovers to Q (acceptable behavior) in time θ and subsequently from each state in Q, it recovers to states in S (ideal behavior) in time δ. We denote this variation as strict 2-phase recovery.

Another variation is $(\neg S \mapsto_{\leq \theta} (Q - S))$ and $(Q \mapsto_{\leq \delta} S)$, i.e., the program first recovers to $(Q - S)$ in time θ and subsequently from each state in Q, it recovers to S in time δ. We denote this variation as ordered-strict 2-phase recovery. One motivation for such a requirement is that we first *record* the occurrence of the fault before ideal behavior can resume. Thus, the program behavior while *recording* the fault (e.g., notifying the user) is strictly different from its ideal behavior.

Third possible variation is $(\neg S \mapsto_{\leq \theta} Q)$ and $(\neg S \mapsto_{\leq \delta} S)$, i.e., the program recovers to Q (acceptable behavior) in time θ and it recovers to states in S (ideal behavior) in time δ. We denote this variation as relaxed 2-phase recovery. One motivation for such requirements is to provide a tradeoff for the designer. In particular, if one can obtain a quick recovery to Q, then one can utilize the remaining time budget in recovering to S. Observe that such tradeoff is not possible in strict 2-phase recovery.

Fourth possible variation is $(Q \mapsto_{\leq \delta} S)$ and $(\neg S \mapsto_{\leq \theta} S)$, i.e., if the program is perturbed to Q, then it recovers to S in time δ and if the program is perturbed to any state, then it recovers to a state in S in time θ. We denote this variation as graceful 2-phase recovery. One motivation for such requirements is a scenario where (1) faults that perturb the program to Q only are more common and, hence, a quick recovery (small δ) is desirable in restoring the ideal behavior, and (2) faults that perturb the program to $\neg Q$ are rare and, hence, slow recovery (large θ) is permissible.

In [10], we introduced the notion of bounded-time 2-phase recovery in a general sense. We also addressed the complexity of synthesizing fault-tolerant real-time programs that mask the occurrence of faults and provide strict and ordered-strict 2-phase recovery. In this paper, we focus on synthesis of relaxed and graceful 2-phase recovery. The main contributions of the paper are as follows:

- We formally define and classify different types of bounded-time 2-phase recovery in the context of fault-tolerant real-time programs.
- Regarding synthesizing relaxed 2-phase recovery, we show that
  - the general problem is NP-complete,

  - the problem can be solved in polynomial-time, if $S \subseteq Q$ and it is required that Q be closed, i.e., the program cannot begin in a state in Q and reach a state outside Q, and
  - the problem remains NP-complete, if $S \subseteq Q$ but Q is not required to be closed.
- Regarding synthesizing graceful 2-phase recovery, we show that the problem can always be solved in polynomial-time.

We emphasize that all complexity results are in the size of the input program's region graph.
Organization of the paper. The rest of the paper is organized as follows. Section 2 is dedicated to define real-time programs and specifications. In Section 3, we formally define the different variations of 2-phase recovery. In Section 4, we define the problem statement for synthesizing 2-phase recovery. Section 5 presents our results on the complexity of synthesis of relaxed and graceful 2-phase recovery. We illustrate the application of our algorithms by synthesizing examples in Section 6. Finally, we conclude in Section 7.

2 Real-Time Programs and Specifications

In our framework, real-time programs are specified in terms of their state space and their transitions [3, 2]. The definition of specification is adapted from Alpern and Schneider [1] and Henzinger [17].

2.1 Real-Time Program

Let $V = \{v_1, v_2 \cdots v_n\}$, $n \geq 1$, be a finite set of *discrete variables* and $X = \{x_1, x_2 \cdots x_m\}$, $m \geq 1$, be a finite set of *clock variables*. Each discrete variable v_i, $1 \leq i \leq n$, is associated with a finite *domain* D_i of values. Each clock variable x_j, $1 \leq j \leq m$, ranges over nonnegative real numbers (denoted $\mathbb{R}_{\geq 0}$). A *location* is a function that maps discrete variables to a value from their respective domain. A *clock constraint* over the set X of clock variables is a Boolean combination of formulae of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in X$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over X by $\Phi(X)$. A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable.

For $\tau \in \mathbb{R}_{\geq 0}$, we write $\nu + \tau$ to denote $\nu(x) + \tau$ for every clock variable x in X. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clock variables in X. A *state* (denoted σ) is a pair (s, ν), where s is a location and ν is a clock valuation for X. Let u be a (discrete or clock) variable and σ be a state. We denote the value of u in state σ by $u(\sigma)$. A *transition* is an ordered pair (σ_0, σ_1), where σ_0 and σ_1 are two states. Transitions are classified into two types:

- *Immediate transitions:* $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$, where s_0 and s_1 are two locations, ν is a clock valuation, and λ is a set of clock variables, where $\lambda \subseteq X$.

– *Delay transitions:* $(s, \nu) \rightarrow (s, \nu + \delta)$, where s is a location, ν is a clock valuation, and $\delta \in \mathbb{R}_{\geq 0}$ is a *time duration*. Note that a delay transition only advances time and does not change the location. We denote a delay transition of duration δ at state σ by (σ, δ).

Thus, if ψ is a set of transitions, we let ψ^s and ψ^d denote the set of immediate and delay transitions in ψ, respectively.

Definition 1 (real-time program) A *real-time program* $\mathcal{P}$ is a tuple $\langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$, where $S_\mathcal{P}$ is the *state space* (i.e., the set of all possible states), and $\psi_\mathcal{P}$ is a set of transitions such that $\psi_\mathcal{P} \subseteq S_\mathcal{P} \times S_\mathcal{P}$. ∎

Definition 2 (state predicate) Let $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ be a real-time program. A *state predicate* S of program $\mathcal{P}$ is any subset of $S_\mathcal{P}$, such that if φ is a constraint involving clock variables in X, where $S \Rightarrow \varphi$, then $\varphi \in \Phi(X)$, i.e., clock variables are only compared with nonnegative integers. ∎

By *closure* of a state predicate S in a set $\psi_\mathcal{P}$ of transitions, we mean that (1) if an immediate transition originates in S then it must terminate in S, and (2) if a delay transition with duration δ originates in S then it must remain in S continuously, i.e., intermediate states where the delay is in interval $(0, \delta]$ are all in S.

Definition 3 (closure) A state predicate S is *closed* in program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ (or briefly $\psi_\mathcal{P}$) iff

$$(\forall (\sigma_0, \sigma_1) \in \psi^s_\mathcal{P} \; : \; ((\sigma_0 \in S) \Rightarrow (\sigma_1 \in S))) \quad \wedge$$
$$(\forall (\sigma, \delta) \in \psi^d_\mathcal{P} \; : \; ((\sigma \in S) \Rightarrow \forall \epsilon \mid ((\epsilon \in \mathbb{R}_{\geq 0}) \; \wedge \; (\epsilon \leq \delta)) \; : \; \sigma + \epsilon \in S)). \blacksquare$$

Definition 4 (computation) A *computation* of $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ (or briefly $\psi_\mathcal{P}$) is a finite or infinite timed state sequence of the form:

$$\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$$

iff the following conditions are satisfied: (1) $\forall j \in \mathbb{Z}_{\geq 0} : (\sigma_j, \sigma_{j+1}) \in \psi_\mathcal{P}$, (2) if $\overline{\sigma}$ reaches a terminating state σ_f where there does not exist a state σ such that $(\sigma_f, \sigma) \in \psi^s_\mathcal{P}$, then we let $\overline{\sigma}$ stutter at σ_f, but advance time indefinitely, and (3) the sequence $\tau_0, \tau_1, \cdots$ (called the *global time*), where $\tau_i \in \mathbb{R}_{\geq 0}$ for all $i \in \mathbb{Z}_{\geq 0}$, satisfies the following constraints:

1. *(monotonicity)* for all $i \in \mathbb{Z}_{\geq 0}$, $\tau_i \leq \tau_{i+1}$,
2. *(divergence)* if $\overline{\sigma}$ is infinite, for all $t \in \mathbb{R}_{\geq 0}$, there exists $j \in \mathbb{Z}_{\geq 0}$ such that $\tau_j \geq t$, and
3. *(consistency)* for all $i \in \mathbb{Z}_{\geq 0}$, (1) if (σ_i, σ_{i+1}) is a delay transition (σ_i, δ) in $\psi^d_\mathcal{P}$ then $\tau_{i+1} - \tau_i = \delta$, and (2) if (σ_i, σ_{i+1}) is an immediate transition in $\psi^s_\mathcal{P}$ then $\tau_i = \tau_{i+1}$. ∎

We distinguish between a *terminating* computation and a *deadlocked* finite computation. Precisely, when a computation $\overline{\sigma}$ terminates in state σ_f, we include the delay transitions (σ_f, δ) in $\psi^d_{\mathcal{P}}$ for all $\delta \in \mathbb{R}_{\geq 0}$, i.e., $\overline{\sigma}$ can be extended to an infinite computation by advancing time arbitrarily. On the other hand, if there exists a state σ_d, such that there is no outgoing (delay or immediate) transition from σ_d then σ_d is a *deadlock state*.

2.2 Specification

Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ be a program. A *specification* (or *property*), denoted *SPEC*, for $\mathcal{P}$ is a set of infinite computations of the form $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ where $\sigma_i \in S_{\mathcal{P}}$ for all $i \in \mathbb{Z}_{\geq 0}$. Following Henzinger [17], we require that all computations in *SPEC* satisfy time-monotonicity and divergence. We now define what it means for a program to satisfy a specification.

Definition 5 (satisfies) Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ be a program, S be a state predicate, and *SPEC* be a specification for $\mathcal{P}$. We write $\mathcal{P} \models_S$ *SPEC* and say that *$\mathcal{P}$ satisfies SPEC from S* iff (1) S is closed in $\psi_{\mathcal{P}}$, and (2) every computation of $\mathcal{P}$ that starts from S is in *SPEC*. ∎

Definition 6 (invariant) Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ be a program, S be a state predicate, and *SPEC* be a specification for $\mathcal{P}$. If $\mathcal{P} \models_S$ *SPEC* and $S \neq \{\}$, we say that S is an *invariant of $\mathcal{P}$ for SPEC*. ∎

Whenever the specification is clear from the context, we will omit it; thus, "S is an invariant of $\mathcal{P}$" abbreviates "S is an invariant of $\mathcal{P}$ for *SPEC*". Note that Definition 5 introduces the notion of satisfaction with respect to infinite computations. In case of finite computations, we characterize them by determining whether they can be extended to an infinite computation in the specification.

Definition 7 (maintains) We say that program $\mathcal{P}$ *maintains SPEC* from S iff (1) S is closed in $\psi_{\mathcal{P}}$, and (2) for all computation prefixes $\overline{\alpha}$ of $\mathcal{P}$ that start in S, there exists a computation suffix $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta} \in$ *SPEC*. We say that $\mathcal{P}$ *violates SPEC* from S iff it is not the case that $\mathcal{P}$ maintains *SPEC* from S. ∎

We note that if $\mathcal{P}$ satisfies *SPEC* from S then $\mathcal{P}$ maintains *SPEC* from S as well, but the reverse direction does not always hold. We, in particular, introduce the notion of *maintains* for computations that a (fault-intolerant) program cannot produce, but the computation can be extended to one that is in *SPEC* by adding *recovery* computation suffixes, i.e., $\overline{\alpha}$ may be a computation prefix that leaves S, but $\overline{\beta}$ brings the program back to S (see Section 3 for details).

Specifying timing constraints. In order to express time-related behaviors of real-time programs (e.g., deadlines and recovery time), we focus on a standard property typically used in real-time computing known as the *bounded response property*. A bounded response property, denoted $P \mapsto_{\leq \delta} Q$ where P and Q are two state predicates and $\delta \in \mathbb{Z}_{\geq 0}$, is the set of all computations $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ in which, for all $i \geq 0$, if $\sigma_i \in P$ then there exists j, $j \geq i$, such that (1) $\sigma_j \in Q$, and (2) $\tau_j - \tau_i \leq \delta$, i.e., it is always the case that a state in P is followed by a state in Q within δ time units.

The specifications considered in this paper are an intersection of a *safety* specification and a *liveness* specification [1, 17]. In this paper, we consider a special case where safety specification is characterized by a set of bad immediate transitions and a set of bounded response properties.

Definition 8 (safety specification) Let *SPEC* be a specification. The *safety specification* of *SPEC* is the union of the sets $SPEC_{\overline{bt}}$ and $SPEC_{\overline{br}}$ defined as follows:

1. *(timing-independent safety)* Let $SPEC_{bt}$ be a set of immediate *bad transitions*. We denote the specification whose computations have no transition in $SPEC_{bt}$ by $SPEC_{\overline{bt}}$.
2. *(timing constraints)* We denote $SPEC_{\overline{br}}$ by the conjunction $\bigwedge_{i=0}^{m}(P_i \mapsto_{\leq \delta_i} Q_i)$, for state predicates P_i and Q_i, and, response times δ_i. ∎

Throughout the paper, $SPEC_{\overline{br}}$ is meant to prescribe how a program should carry out bounded-time phased recovery to its normal behavior after the occurrence of faults. We formally define the notion of recovery in Section 3.

Definition 9 (liveness specification) A liveness specification of *SPEC* is a set of computations that meets the following condition: for each finite computation $\overline{\alpha}$, there exists a computation $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta} \in SPEC$. ∎

Remark 1. In our synthesis problem in Section 4, we begin with an initial program that satisfies its specification (including the liveness specification). We will show that our synthesis techniques *preserve* the liveness specification. Hence, the liveness specification need not be specified explicitly. ∎

3 Fault Model and Fault-Tolerance

3.1 Fault Model

The faults that a program is subject to are systematically represented by transitions. A class of *faults* f for program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ is a subset of *immediate* and *delay* transitions of the set $S_{\mathcal{P}} \times S_{\mathcal{P}}$. We use $\psi_{\mathcal{P}}[]f$ to denote the transitions obtained by taking the union of the transitions in $\psi_{\mathcal{P}}$ and the transitions in f. We emphasize that such representation is possible for different types of faults (e.g., stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults.

Definition 10 (fault-span) We say that a state predicate T is an f-span (read as *fault-span*) of $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ from S iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $\psi_{\mathcal{P}}[]f$. ∎

Observe that for all computations of $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ that start from states where S is true, T is a boundary in the state space of $\mathcal{P}$ up to which (but not beyond which) the state of $\mathcal{P}$ may be perturbed by the occurrence of the transitions in f. Subsequently, as we defined the computations of $\mathcal{P}$, one can define computations of program $\mathcal{P}$ in the presence of faults f by simply substituting $\psi_\mathcal{P}$ with $\psi_\mathcal{P}[]f$ in Definition 4.

3.2 Phased Recovery and Fault-Tolerance

Now, we define the different types of 2-phase recovery properties discussed in Section 1.

Definition 11 (2-phase recovery) Let $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ be a real-time program with invariant S, Q be an arbitrary *intermediate recovery predicate*, f be a set of faults, and *SPEC* be a specification (as defined in Definitions 8 and 9). We say that $\mathcal{P}$ *provides* (ordered-strict, strict, relaxed or graceful) 2-phase recovery from S and Q with recovery times $\delta, \theta \in \mathbb{Z}_{\geq 0}$, respectively, iff $\langle S_\mathcal{P}, \psi_\mathcal{P}[]f \rangle$ maintains $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q_1) \ \wedge \ (Q_2 \mapsto_{\leq \delta} S)$ from S, where, depending upon the type of the desired 2-phase recovery, Q_1 and Q_2 are instantiated as follows:

	ordered-strict	strict	relaxed	graceful
Q_1	$Q - S$	Q	Q	S
Q_2	Q	Q	$\neg S$	Q

We call θ and δ *intermediate recovery time* and *recovery time*, respectively. ∎

Definition 12 (fault-tolerance) Let $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ be a real-time program with invariant S, f be a set of faults, and *SPEC* be a specification as defined in Definitions 8 and 9. We say that $\mathcal{P}$ is *f-tolerant to SPEC from S*, iff (1) $\mathcal{P} \models_S SPEC$, and (2) there exists T such that T is an f-span of $\mathcal{P}$ from S and $\langle S_\mathcal{P}, \psi_\mathcal{P}[]f \rangle$ maintains *SPEC* from T. ∎

Notation. Whenever the specification *SPEC* and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant to *SPEC* from S".

4 Problem Statement

Given are a fault-intolerant real-time program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$, its invariant S, a set f of faults, and a specification *SPEC* such that $\mathcal{P} \models_S SPEC$. Our goal is to synthesize a real-time program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ with invariant S' such that $\mathcal{P}'$ is f-tolerant to *SPEC* from S'. We require that our synthesis methods obtain $\mathcal{P}'$ from $\mathcal{P}$ by *adding fault-tolerance* to $\mathcal{P}$ without introducing new behaviors in the absence of faults. To this end, we first define the notion of *projection*. Projection of a set $\psi_\mathcal{P}$ of transitions on state predicate S consists of immediate transitions of $\psi_\mathcal{P}^s$ that start in S and end in S, and delay transitions of $\psi_\mathcal{P}^d$ that start and remain in S continuously.

Definition 13 (projection) *Projection* of a set ψ of transitions on a state predicate S (denoted $\psi|S$) is the following set of transitions:

$$\begin{aligned}\psi|S = &\{(\sigma_0, \sigma_1) \in \psi^s \mid \sigma_0, \sigma_1 \in S\} \ \cup \\ &\{(\sigma, \delta) \in \psi^d \mid \sigma \in S \ \wedge \ (\forall \epsilon \mid ((\epsilon \in \mathbb{R}_{\geq 0}) \ \wedge \ (\epsilon \leq \delta)) \ : \ \sigma + \epsilon \in S)\}. \ \blacksquare\end{aligned}$$

Since meeting timing constraints in the presence of faults requires time predictability, we let our synthesis methods incorporate a finite set Y of new clock variables. We denote the set of states obtained by abstracting the clock variables in Y from a state predicate U by $U \backslash Y$. Likewise, if ψ is a set of transitions, we denote the set of transitions obtained by abstracting the clock variables in Y by $\psi_{\mathcal{P}} \backslash Y$. Now, observe that in the absence of faults, if S' contains states that are not in S then $\mathcal{P}'$ may include computations that start outside S. Hence, we require that $(S' \backslash Y) \subseteq S$. Moreover, if $\psi'_{\mathcal{P}}|S'$ contains a transition that is not in $\psi_{\mathcal{P}}|S'$ then in the absence of faults, $\mathcal{P}'$ can exhibit computations that do not correspond to computations of $\mathcal{P}$. Therefore, we require that $(\psi_{\mathcal{P}'} \backslash Y)|(S' \backslash Y) \subseteq \psi_{\mathcal{P}}|(S' \backslash Y)$.

Problem Statement 1 Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$, invariant S, specification SPEC, and set f of faults such that $\mathcal{P} \models_S \mathit{SPEC}$, identify $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ and S' such that:

$(C1)$ $S_{\mathcal{P}'} \backslash Y = S_{\mathcal{P}}$, where Y is a finite set of new clock variables,
$(C2)$ $(S' \backslash Y) \subseteq S$,
$(C3)$ $(\psi_{\mathcal{P}'} \backslash Y) \mid (S' \backslash Y) \subseteq \psi_{\mathcal{P}}|(S' \backslash Y)$, and
$(C4)$ $\mathcal{P}'$ is f-tolerant to SPEC from S'. $\blacksquare$

Note that the above problem statement can be instantiated for all four types of 2-phase recovery. In this paper, we focus on relaxed and graceful 2-phase recovery. Hence, we instantiate the problem statement with these two types and whenever it is clear from the context, for brevity, we omit the instantiation.

Notice that conditions $C1..C3$ in Problem Statement 1 precisely express the notion of behavior restriction (also called *language inclusion*) used in controller synthesis and game theory. Moreover, constraint $C4$ implicitly implies that the synthesized program is not allowed to exhibit new finite computations, which is known as the *non-blocking* condition. It is easy to observe that unlike controller synthesis problems, our notion of *maintains* (cf. Definition 7) embedded in condition $C4$ allows the output program to exhibit recovery computations that input program does not have.

5 Synthesizing Relaxed and Graceful 2-Phase Recovery

In this section, first, in Subsection 5.1, we show that the problem of synthesizing relaxed 2-phase recovery is NP-complete. Then, in Subsection 5.2, we show that it can be solved in polynomial-time if Q is required to be closed in the synthesized program. Subsequently, we interpret this result and identify its effect in Subsection 5.3. We present our polynomial algorithm for graceful 2-phase recovery in Subsection 5.4. Finally, we consider whether there are other types of 2-phase recovery instances in Subsection 5.5.

5.1 Complexity of Synthesizing Relaxed 2-Phase Recovery

In this section, we show that, in general, the problem of synthesizing fault-tolerant real-time programs that provide relaxed 2-phase recovery is NP-complete in the size of locations of the given fault-intolerant real-time program.

Instance. A real-time program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ with invariant S, a set of faults f, and a specification $SPEC$, such that $\mathcal{P} \models_S SPEC$, where $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \wedge (\neg S \mapsto_{\leq \delta} S)$ for state predicate Q and $\delta, \theta \in \mathbb{Z}_{\geq 0}$.

The decision problem (R2P). Does there exist an f-tolerant program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ with invariant S' such that $\mathcal{P}'$ and S' meet the constraints of Problem Statement 1 when instantiated with relaxed 2-phase recovery?

We now show that the R2P problem is NP-complete by reducing the *2-path problem* [16] to R2P.

The simplified 2-path problem (2PP). Given are a digraph $G = \langle V, A \rangle$, where V is a set of vertices and A is a set of arcs, and three distinct vertices $v_1, v_2, v_3 \in V$. Decide whether G has a simple (v_1, v_3)-path that also contains vertex v_2 [6].

Theorem 1. *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* relaxed *2-phased recovery is NP-complete in the size of locations of the fault-intolerant program.*

Proof. Since proving membership to NP is trivial, we only show that the problem is NP-hard.

Mapping. Given an instance of the 2PP problem (i.e., $G = \langle V, A \rangle$, v_1, v_2, and v_3), we first present a polynomial-time mapping from the 2PP instance to an instance of the R2P problem (i.e., $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$, S, f, and $SPEC$) as follows (see also Figure 1):

- (*clock variables*) $X = \{x\}$.
- (*locations*) $loc_\mathcal{P} = \{s_v \mid v \in V\} \cup \{s_l\}$.
- (*state space*) $S_\mathcal{P} = \{(s, \nu) | s \in loc_\mathcal{P} \wedge \nu(x) \geq 0\}$.
- (*program transitions*) $\psi_\mathcal{P} =$
$$\{(s_u, x = 1) \rightarrow (s_v, x := 0) \mid (u, v) \in A \wedge u \neq v_3\} \cup$$
$$\{(s_{v_3}, x = |N| + 1) \rightarrow (s_l, x := 0)\} \cup$$
$$\{(s_l, x = 1) \rightarrow (s_l, x := 0)\}.$$
- (*invariant*) $S = \{(s_l, \nu) \mid \nu(x) \leq 1\}$.
- (*fault transitions*) $f = \{(s_l, x \geq 0) \rightarrow (s_{v_1}, x := 0)\}$.
- (*safety specification*) $SPEC_{bt} = S_\mathcal{P} \times S_\mathcal{P} - (\psi_\mathcal{P} \cup f)$ and
$$SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \wedge (\neg S \mapsto_{\leq \delta} S),$$
where $Q = \{(s_{v_2}, \nu_1), (s_l, \nu_2) \mid \nu_1(x), \nu_2(x) \leq 1\}$, $\delta = \infty$, and $\theta = |N|$.

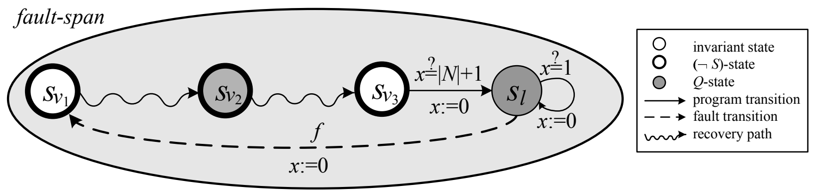

Fig. 1. Mapping 2-path problem to synthesizing relaxed 2-phase recovery.

An intuitive description of the above mapping is as follows. First, we include one clock variable x in $\mathcal{P}$. The locations of $\mathcal{P}$ consists of all vertices in V and an additional vertex s_l. The state space of the program is obtained by considering all possible values in $\mathbb{R}_{\geq 0}$ for x at any location. The program invariant S merely includes states where the location is s_l. The set of program transitions consists of all arcs in A except arcs that originate at v_3, a transition from s_{v_3} to s_l and a self-loop at s_l. Inclusion of the self-loop guarantees that all program computations are infinite. Exclusion of arcs that originate from s_l ensures the closure of S. Furthermore, the delay on the transitions corresponding to the original graph is 1. And, the delay on the transition from s_{v_3} to s_l is $|N|+1$. Hence, to meet the timing constraint $(\neg S \mapsto_{\leq \theta} Q)$, the program must reach s_{v_2} and not s_l. Finally, we let $SPEC_{bt}$ be the set of all transitions except those identified above. Thus, the program can use all transitions corresponding to the original graph but no other (new) transitions, as they would violate timing independent safety.

Reduction. Given the above mapping, we now show that 2PP has a solution iff the answer to the R2P problem is affirmative:

– $(\Rightarrow)$ Let the answer to 2PP be a simple path Π that originates at v_1, ends at v_3, and contains v_2. We claim that in the structure shown in Figure 1, the set of program transitions $\psi_{\mathcal{P}'}$ obtained by taking only the transitions corresponding to the arcs along Π, plus the transition $(s_{v_3}, x = |N|+1) \rightarrow (s_l, x := 0)$, and the self-loop at s_l satisfies the constraints of Problem Statement 1 when instantiated with relaxed 2-phase recovery. We prove our claim as follows. Notice that (1) $S_{\mathcal{P}} = S_{\mathcal{P}'}$, (2) $S' = S = \{s_l\}$, (3) $\psi_{\mathcal{P}'}|S' \subseteq \psi_{\mathcal{P}}|S'$, and (4) $\mathcal{P}'$ is fault-tolerant to $SPEC$ from S', as (i) in the absence of faults, by starting from the invariant S', all computations of $\psi_{\mathcal{P}'}$ are infinite, and (ii) in the presence of faults, $\mathcal{P}' \models_{S'} SPEC_{\overline{br}}$ (since Π is a simple path that reaches Q and S in desired timing constraints, respectively), and, $\mathcal{P}' \models_{S'} SPEC_{\overline{bt}}$.

– $(\Leftarrow)$ Let the answer to the R2P problem be $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ with invariant S'. Since S' must be nonempty, $S' = \{(s_l, \nu) | \nu(x) \leq 1\}$. Now, consider a computation prefix of $\mathcal{P}'$ that starts from S' and the fault transition (s_l, s_{v_1}) perturbs the state of $\mathcal{P}'$. Since $\mathcal{P}'$ is fault-tolerant it must satisfy the bounded response properties $\neg S' \mapsto_{\leq \theta} Q$ and $\neg S \mapsto_{\leq \delta} S'$. Hence, there should exist a computation prefix $\overline{\sigma}$ that originates at $\{s_{v_1}\}$ and reaches $Q = \{s_{v_2}, s_l\}$. However, based on the above construction reaching s_l within time θ is impossible. Hence, $\overline{\sigma}$ must reach s_{v_2}. Moreover, $\overline{\sigma}$ must also visit $S' = \{s_l\}$. To this end, based on the above construction, $\overline{\sigma}$ must also reach s_{v_3}. Since there is only a self-loop transition in s_l, starting from s_{v_1}, $\overline{\sigma}$ must first visit s_{v_2}, then s_{v_3} and finally s_l. Let $\overline{\sigma}_p$ be the prefix of $\overline{\sigma}$ that terminates in s_{v_3}. Now, based on the above observations the path, say Π, whose vertices and arcs correspond to state and transitions in $\overline{\sigma}_p$, is the answer to 2PP. ∎

Observe that based on the above proof, the problem remains NP-complete even if the instance of R2P satisfies the constraint $S \subseteq Q$.

Corollary 1. *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* relaxed *2-phased recovery is NP-complete in the size of locations of the fault-intolerant program even if $S \subseteq Q$.* ∎

5.2 Synthesizing Relaxed 2-Phase Recovery with Closure of Q

In this section, we show that if the intermediate predicate Q is required to be closed in the synthesized program, then the problem of synthesizing relaxed 2-phase recovery can be solved in polynomial time in the size of the time-abstract bisimulation of the input program. Towards this end, we propose the algorithm Add_RelaxedPhasedRecovery .

Assumption 1 For simplicity of presentation, we assume that the number of fault occurrences in any computation is at most 1. Note that the proof of Theorem 1 is valid even with this assumption. In [8], we have shown that if multiple faults occur within a computation, for a given state, one can compute the maximum time required to reach a state predicate. ∎

Our algorithm utilizes region graph [2] representation of the given input program. Intuitively, the region graph allows us to construct a finite representation of the state space of the given real-time program. To illustrate this, consider a program that has a single clock variable, say x. Based on the constraints on the use of clock variables, the program can distinguish between $x = 0$, $0 < x < 1$, $x = 1$, $1 < x < 2$, etc. However, it cannot distinguish between $x = 0.5$ and $x = 0.7$. The regions constructed by the region graph allows one to identify these equivalence regions. Our algorithm only relies on this equivalence property of the regions. However, for reader's convenience, we repeat the relevant details of the region graph construction.

Region Graph. Given a real-time program $\mathcal{P}$, for each clock variable $x \in X$, let c_x be the largest constant in clock constraint of transitions of $\psi_{\mathcal{P}}$ that involve x. We say that two clock valuations ν, μ are *clock equivalent* if (1) for all $x \in X$, either $\lfloor \nu(x) \rfloor = \lfloor \mu(x) \rfloor$ or both $\nu(x), \mu(x) > c_x$, (2) the ordering of the fractional parts of the clock variables in the set $\{x \in X \mid \nu(x) < c_x\}$ is the same in μ and ν, and (3) for all $x \in X$ where $\nu(x) < c_x$, the clock value $\nu(x)$ is an integer iff $\mu(x)$ is an integer. A *clock region* ρ is a clock equivalence class. Two states (s_0, ν_0) and (s_1, ν_1) are region equivalent, written $(s_0, \nu_0) \equiv (s_1, \nu_1)$, if (1) $s_0 = s_1$, and (2) ν_0 and ν_1 are clock equivalent. A *region* $r = (s, \rho)$ is an equivalence class with respect to $\equiv$, where s is a location and ρ is a clock region. We say that a

Algorithm 1 Add_RelaxedPhasedRecovery

Input: A real-time program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ with invariant S, fault transitions f, bad transitions $SPEC_{bt}$, and $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \wedge (\neg S \mapsto_{\leq \delta} S)$, where Q is an intermediate recovery predicate, such that $S \subseteq Q$.
Output: If successful, a fault-tolerant real-time program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ and invariant S' such that $\langle S_\mathcal{P}, \psi'_\mathcal{P} [] f \rangle \models_{S'} SPEC_{\overline{br}}$ and Q is closed in $\psi_{\mathcal{P}'}$.

```
1: $\langle S^r_\mathcal{P}, \psi^r_\mathcal{P} \rangle,\ S^r_1,\ Q^r,\ f^r,\ SPEC^r_{bt} :=$ ConstructRegionGraph($\langle S_\mathcal{P}, \psi_\mathcal{P} \rangle,\ S,\ Q,\ f,\ SPEC_{bt}$);
2: $ms\ :=\ \{r_0 \mid \exists r_1, r_2 \cdots r_n\ :\ (\forall j \mid 0 \leq j < n : (r_j, r_{j+1}) \in f^r)\ \wedge\ (r_{n-1}, r_n) \in SPEC^r_{bt}\}$;
3: $mt\ :=\ \{(r_0, r_1) \in \psi^r_\mathcal{P} \mid (r_1 \in ms)\ \vee\ ((r_0, r_1) \in SPEC^r_{bt})\}$;
4: $T^r_1\ :=\ S^r_\mathcal{P} - ms$;
5: repeat
6:     $T^r_2,\ S^r_2\ :=\ T^r_1,\ S^r_1$;
7:     $\psi^r_{\mathcal{P}_1}\ :=\ \psi^r_\mathcal{P}|S^r_1\ \cup\ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (T^r_1 - Q^r)\ \wedge\ (s_1, \rho_1) \in T^r_1\ \wedge$
                 $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0\ :\ (\exists \lambda \subseteq X\ :\ \rho_1 = \rho_2[\lambda := 0])\}\ \cup$
                         $\{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (Q^r - S^r_1)\ \wedge\ (s_1, \rho_1) \in Q^r\ \wedge$
                 $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0\ :\ (\exists \lambda \subseteq X\ :\ \rho_1 = \rho_2[\lambda := 0])\}\ -\ mt$;
8:     $\psi^r_{\mathcal{P}_1},\ ns\ :=$ Add_BoundedResponse($\langle S^r_\mathcal{P}, \psi^r_{\mathcal{P}_1} \rangle,\ Q^r - S^r,\ S^r,\ \delta$);
9:     $T^r_1\ :=\ T^r_1\ -\ ns$;
10:    $\psi^r_{\mathcal{P}_1},\ ns\ :=\ \overline{transform}$ (Add_BoundedResponse($transform(\langle S^r_\mathcal{P}, \psi^r_{\mathcal{P}_1} \rangle),\ T^r_1 - Q^r,\ Q^r,\ \theta$));
11:    $T^r_1,\ Q^r\ :=\ T^r_1\ -\ ns,\ Q^r\ -\ ns$;
12:    while $(\exists r_0, r_1 : r_0 \in T^r_1\ \wedge\ r_1 \notin T^r_1\ \wedge\ (r_0, r_1) \in f^r)$ do
13:        $T^r_1 := T^r_1 - \{r_0\}$;
14:    end while
15:    while $(\exists r_0 \in (S^r_1 \cap T^r_1)\ :\ (\forall r_1 \mid (r_1 \neq r_0\ \wedge\ r_1 \in S^r_1)\ :\ (r_0, r_1)\ \notin\ \psi^r_{\mathcal{P}_1}))$ do
16:        $S^r_1 := S^r_1 - \{r_0\}$;
17:    end while
18:    if $(S^r_1 = \{\}\ \vee\ T^r_1 = \{\})$ then
19:        print ``no solution to relaxed 2-phase recovery exists''; exit;
20:    end if
21: until $(T_1 = T_2\ \wedge\ S_1 = S_2)$
22: $\langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle,\ S', T'\ :=$ ConstructRealTimeProgram($\langle S^r_\mathcal{P}, \psi^r_{\mathcal{P}_1} \rangle,\ S^r_1,\ T^r_1$);
23: return $\langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle,\ S',\ T'$;
```

clock region β is a *time-successor* of a clock region α iff for each $\nu \in \alpha$, there exists $\tau \in \mathbb{R}_{\geq 0}$, such that $\nu + \tau \in \beta$, and $\nu + \tau' \in \alpha \cup \beta$ for all $\tau' < \tau$.

Next, we describe our algorithm Add_RelaxedPhasedRecovery:

– (*Step 1: Initialization*) Using the technique described above from [2], we obtain the region graph, $R(\mathcal{P})$, for the input program by using the routine ConstructRegionGraph (Line 1). Vertices of $R(\mathcal{P})$ (denoted $S^r_\mathcal{P}$) are regions. Edges of $R(\mathcal{P})$ (denoted $\psi^r_\mathcal{P}$) are of the form $(s_0, \rho_0) \to (s_1, \rho_1)$ iff for some clock valuations $\nu_0 \in \rho_0$ and $\nu_1 \in \rho_1$, $(s_0, \nu_0) \to (s_1, \nu_1)$ is a transitions in $\psi_\mathcal{P}$. Likewise, we convert state predicates and sets of transitions into corresponding region predicates and sets of edges. For example, S^r denotes the region predicate obtained from input S, and it is obtained as $S^r = \{(s, \rho) \mid \exists (s, \nu) : ((s, \nu) \in S\ \wedge\ \nu \in \rho)\}$.
  In order to ensure that the synthesized program does not violate timing-independent safety, we identify the set ms of regions from where a computation can violate $SPEC_{\overline{bt}}$ by the occurrence of faults alone (Line 2). Clearly, the program should not reach a region in ms. Hence, we remove (Line 4) ms from the region predicate T^r_1, which is used to compute the fault-span of the program being synthesized. The set of edges that should not be included in the synthesized program, mt, consists of edges that reach ms and the

edges in $SPEC_{bt}^{r}$. These edges are removed while constructing the possible program transitions (Line 7).

- (*Step 2: Adding* $(Q \mapsto_{\leq \delta} S)$) In this step, we first recompute the set $\psi_{\mathcal{P}_1}$ of program edges (Line 7) that could potentially be used during phased recovery in the synthesized program. Towards this end, we partition the edges based on their originating states: If an edge originates from a state in S_1^r (estimated invariant of the synthesized program), then by constraint $C3$ of Problem Statement 1, the edge must be included in the original program. If the edge originates in a region in $Q_1^r - S_1^r$ then due to closure requirement of Q, it must end in Q_1^r. And, if the edge originates in a region in $T_1^r - Q_1^r$ then due to closure requirement of fault-span, it must end in T_1^r. Furthermore, the edges must meet the time monotonicity condition and not present in the set mt. It is straightforward to observe that the edges computed on Line 7 must be a superset of the edges in a program that satisfies constraints of Problem Statement 1.
  We use the program constructed on Line 7 and invoke the procedure Add_BoundedResponse to add $(Q \mapsto_{\leq \delta} S)$. Add_BoundedResponse (from [7]) adds a clock variable, say t_1, which gets reset when $Q - S$ becomes true. It computes a shortest path from every region in $Q_1^r - S_1^r$ to some region in S_1^r. If the delay on this path is less than or equal to δ, it includes that path in the synthesized program. If the delay is more than δ then it includes the corresponding region in $Q_1^r - S_1^r$ in ns. It follows that regions in ns cannot be included in the synthesized program. Hence, we remove ns from T_1^r and Q^r on Lines 9 and 11, respectively. Add_BoundedResponse can also add additional paths whose length is larger than that of the shortest paths but less than δ. However, for relaxed 2-phase recovery, addition of such additional paths needs to be performed after adding the second timing constraint in Line 10.
- (*Step 3: Adding* $(\neg S \mapsto_{\leq \gamma} Q)$) For each region r in Q^r, we identify $wt(r)$ that denotes the length of the path from r to a region in S^r. Next, we add the property $(\neg S \mapsto_{\leq \gamma} Q)$, where the value of γ depends upon the exact state reached in Q. Since we need to ensure $(\neg S \mapsto_{\leq \theta} Q)$, γ must be less than θ. And, since we need to ensure $(\neg S \mapsto_{\leq \delta} S)$, the time to reach a region r in Q^r must be less than $\delta - wt(r)$.
  To achieve this with Add_BoundedResponse, we transform the given region graph by the function *transform*, where we replace each region r in Q^r by r_1 (that is outside Q^r) and r_2 (that is in Q^r) such that there is an edge from r_1 to r_2. All incoming edges from $T_1^r - Q^r$ to r now reach r_1. All other edges (edges reaching r from another state in Q^r and outgoing edges from r) are connected to r_2. The weight of the edge from r_1 to r_2 is set to $\max(0, \theta + wt(r) - \delta)$. Now, we call Add_BoundedResponse add $(T_1 - Q \mapsto_{\leq \theta} Q)$. Notice that the transformation of the region graph along with invocation of Add_BoundedResponse (Line 10) ensures that any computation of the synthesized program that that starts from a state σ_0 in $\neg S$ and reaches a state σ_1 in $Q - S$ within θ still has sufficient time to reach a state σ_2 in S such that the overall delay between σ_0 and σ_2 is less than δ. In other words,

the output program will satisfy $(T_1 - Q \mapsto_{\leq\delta} S)$ no matter what path it takes to achieve 2-phase recovery. We now collapse region r_1 and r_2 (created by *transform*) to obtain region r. We use $\overline{transform}$ to denote such collapsing.

– (*Step 4: Repeat if needed or construct synthesized program*) Since we remove some regions from T_1^r, we ensure closure of T_1^r in f by the loop on Lines 12-14. Furthermore, due to constraint $C4$ of the problem statement, S_1^r cannot have deadlock regions from where there are no outgoing edges. Hence, on Lines 15-17, we remove such deadlocks. If this removal causes S_1^r or T_1^r to be an empty set then the algorithm declares failure (Line 19).
  Since removal of regions from S_1^r or T_1^r can potentially affect the bounded response properties added on Lines 8 and 10, Steps 2 and 3 may have to be repeated. If no regions are removed (i.e., we reach a fixpoint), then we construct the corresponding real-time program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ on Line 22. Since the construction of the region graph is a bisimulation of the corresponding real-time program, such reverse transformation is possible.

We now show that the algorithm Add_RelaxedPhasedRecovery is *sound*, i.e., the synthesized program satisfies the constraints of Problem Statement 1, and complete, i.e., the algorithm finds a fault-tolerant program provided one exists.

Theorem 2. *The Algorithm Add_RelaxedPhasedRecovery is sound and complete.*

Proof. We distinguish the following two cases:

– Regarding soundness, we show that the algorithm Add_RelaxedPhasedRecovery satisfies the constraints of Problem Statement 1 when instantiated with relaxed 2-phase recovery. Let the algorithm add two clock variables t_1 and t_2 when invoking Add_BoundedResponse (cf. Lines 8 and 10). We proceed as follows:
  1. (*Constraints* $C1..C3$) By construction, correctness of these constraints trivially follows.
  2. (*Constraint* $C4$) We distinguish two subgoals based on the behavior of $\mathcal{P}'$ in the absence and presence of faults:
     • We need to show that in the absence of faults, $\mathcal{P}' \models_{S'} SPEC$. To this end, consider a computation $\overline{\sigma}$ of $\psi_{\mathcal{P}}'$ that starts in S'. Since the values of t_1 and t_2 are irrelevant inside S', from $C1$, $\overline{\sigma}$ starts from a state in S, and from $C2$, $\overline{\sigma}$ is a computation of $\psi_{\mathcal{P}}$. Moreover, since we remove deadlock states from S' (cf. Lines 15-17), if $\overline{\sigma}$ is infinite in $\mathcal{P}$, then it is infinite in $\mathcal{P}'$ as well. It follows that $\overline{\sigma} \in SPEC$. Hence, every computation of $\psi_{\mathcal{P}'}$ that starts from a state in S' is in $SPEC$. Also, by construction, S' is closed in $\psi_{\mathcal{P}'}$. Furthermore, for all open regions in S'^r, say r_0, there exists an outgoing edge, say (r_0, r_1), for some $r_1 \in S'^r$ where $r_0 \neq r_1$. Since the intolerant program exhibits no time-convergent behavior, such an edge can only terminate at a different clock region, which in turn advances time by at least one time unit. This implies that in the absence of faults, our algorithm

does not introduce time-convergent computations (*Zeno* behaviors) to $\mathcal{P}'$ and, hence, $\mathcal{P}' \models_{S'} SPEC$.
  - Notice that by construction, T' is closed in $\psi_{\mathcal{P}'}[]f$ (cf. Lines 12-14). Now, consider a computation $\overline{\sigma} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ of $\psi_{\mathcal{P}'}[]f$ that starts from a state in $T' - S'$. We now show that this computation reaches a state in Q within time θ and reaches a state in S' within time δ. Based on the properties of *transform*, where a region r in Q was partitioned into two regions r_1 and r_2, and the soundness of Add_BoundedResponse, $\overline{\sigma}$ reaches a region r_2 in time θ. Moreover, after collapsing r_1 and r_2 into the region r, the recovery time to r is at most θ. It follows that $\overline{\sigma}$ reaches a state in Q in time θ. Also, the time to reach r_2 on Line 10 is at most θ. Hence, the time to reach r_1 is at most $\theta - (\theta + wt(r) - \delta)$. Hence, after collapsing r_1 and r_2 into r, the maximum delay in reaching r is at most $\delta - wt(r)$. Based on the definition of $wt(r)$, time to reach S' is at most δ.

– The proof of completeness is based on the observation that if any region is removed, then it must be removed. In other words, there is no fault-tolerant program that meets the constraints of Problem Statement 1 when instantiated with relaxed 2-phase recovery and includes that region. For example, in the computation of *ms*, if (σ_0, σ_1) is a fault transition and violates $SPEC_{\overline{bt}}$, then state σ_0 must be removed (i.e., not reached). Likewise, *ms* includes states from where execution of faults alone violates $SPEC_{\overline{bt}}$. Hence, they must be removed. In Line 7, we compute the program that includes all possible transitions that may be used in the final program. Due to constraint $C3$ of the Problem Statement 1, any transition that begins in the invariant must be a transition of the fault-intolerant program. Due to closure requirement of Q in the synthesized program, any transition from $Q - S$ must end in Q. And, due to closure of fault-span, any transition that begins in $T - Q$ must end in T. Thus, the transitions computed in Line 7 are maximal. Furthermore, using the property of Add_BoundedResponse, if any state is removed in spite of considering all possible transitions that could be potentially used, then that state must be removed (i.e., states in *ns*). In other words, states in *ns* must be removed since one of the bounded liveness properties cannot be met from those states.
  Our algorithm declares failure when either the invariant or fault-span of the synthesized program is equal to the empty set. In other words, our algorithm fails to find a solution when all states of the intolerant program are illegitimate with respect to Problem Statement 1. This implies our algorithm declares failure only if the corresponding fault-tolerant program does not exist. Therefore, the algorithm Add_RelaxedPhasedRecovery is complete. ∎

5.3 Interpretation of Closure of Q

One main observation from the results in Subsections 5.1 and 5.2 is that the requirement of 'closure of Q', where Q is the intermediate recovery predicate, appears to play a crucial role in reducing the complexity. Thus, one may pose

Algorithm 2 Add_GracefulPhasedRecovery

```
Input: A real-time program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ with invariant $S$, fault transitions $f$, bad transitions
    $SPEC_{bt}$, and $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} S) \;\wedge\; (\neg Q \mapsto_{\leq \delta} S)$, where $Q$ is an intermediate recovery
    predicate, such that $S \subseteq Q$.
Output: If successful, a fault-tolerant real-time program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ and invariant $S'$ such
    that $\langle S_\mathcal{P}, \psi'_\mathcal{P}[]f \rangle \models_{S'} SPEC_{\overline{br}}$.

    // This algorithm is obtained by changing the following lines from Algorithm 1
    7 : $\psi^r_{\mathcal{P}_1} := \psi^r_\mathcal{P}|S^r_1 \cup \{((s_0,\rho_0),(s_1,\rho_1)) \mid (s_0,\rho_0) \in (T^r_1 - S^r) \;\wedge\; (s_1,\rho_1) \in T^r_1 \;\wedge$
            $\exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \;:\; (\exists \lambda \subseteq X \;:\; \rho_1 = \rho_2[\lambda := 0])\} \;-\; mt$;
    10 : $\psi^r_{\mathcal{P}_1},\; ns := $ Add_BoundedResponse$((\langle S^r_\mathcal{P}, \psi^r_{\mathcal{P}_1} \rangle),\, T^r - S^r_1,\, S^r_1,\, \theta)$;
```

questions on the intuitive implication of this requirement in practice. There are two ways of characterizing the intermediate recovery predicate:

- One characterization is that predicate Q identifies an acceptable behavior of the program. In this case, it is expected that once the program starts exhibiting acceptable behavior, it continues to exhibit acceptable (or ideal) behavior in future. In such a characterization, closure of Q is satisfied.
- Another characterization is that the predicate Q identifies a *special* behavior that does not occur in the absence of faults. This special behavior can include notification or recording of the fault, suspension of normal operation for a certain duration, etc. Thus, in such a characterization, the program reaches Q, then leaves Q and eventually starts exhibiting its ideal behavior. In such a characterization, closure of Q is not satisfied.

The results in this paper shows that the complexity of the former characterization is significantly less than the latter. In other words, requiring that faults be recorded causes a significant growth in the complexity.

5.4 Complexity of Synthesizing Graceful 2-Phase Recovery

In this section, we show a somewhat counter-intuitive result that although the general problem of synthesizing strict and relaxed 2-phase recovery are NP-complete, the synthesis problem for graceful 2-phase recovery can be solved in polynomial-time in the size of the input program's region graph. Towards this end, we present a sound and complete solution to the Problem Statement 1 when instantiated for graceful 2-phase recovery. This algorithm also requires Assumption 1 from Subsection 5.2. Without loss of generality, in this algorithm, we assume that $\delta \leq \theta$. If $\delta > \theta$, then graceful 2-phase recovery corresponds to the requirement $(\neg S \mapsto_{\leq \theta} S)$.

We now describe the algorithm Add_GracefulPhasedRecovery. Since this algorithm reuses most of Add_RelaxedPhasedRecovery, we only identify the differences.

- (*Step 1: Initialization*) This step is identical to that in Algorithm 1 and it constructs the region graph $R(\mathcal{P})$.

- (*Step 2: Adding* $(Q \mapsto_{\leq \delta} S)$) In this step, we add recovery paths to $R(\mathcal{P})$ so that $R(\mathcal{P})$ satisfies $(Q \mapsto_{\leq \delta} S)$. The set of edges used in this step (Line 7) differs from the corresponding step in Add_RelaxedPhasedRecovery. In particular, if an edge originates in Q^r_1, it need not terminate in Q^r_1. This is due to the fact that Q is not necessarily closed in graceful 2-phase recovery. Thus, the transitions computed for $\psi_{\mathcal{P}_1}$ of program edges are as specified on Line 7.
  After adding recovery edges, we invoke the procedure Add_BoundedResponse (Line 8) with parameters $Q^r - S^r$, S^r, and δ to ensure that $R(\mathcal{P})$ indeed satisfies the bounded response property $Q \mapsto_{\leq \delta} S$. Since the value of ns returned by Add_BoundedResponse indicates that there does not exist a computation prefix that maintains the corresponding bounded response property from the regions in ns, in Line 9, the algorithm removes ns from T^r_1.
- (*Step 3: Adding* $(\neg S \mapsto_{\leq \theta} S)$) This task is achieved by calling Add_BoundedResponse, where from each state in $\neg S$, we add a shortest path from that state to a state in S. Note that the paths from states in Q have a delay of at most δ. If such a path does not exist from a state in Q then, in Step 2, that state would have been included in ns and, hence, removed from T^r_1. While the addition of the second bounded response property is possible for graceful 2-phase recovery, for reasons discussed after Theorem 3, it is not possible for relaxed 2-phase recovery.
- (*Step 4*) This step is identical to that in Algorithm 1.

Theorem 3. *The Algorithm Add_GracefulPhasedRecovery is sound and complete.*

Proof. The proof of soundness is similar to that of Theorems 2. In particular, regarding soundness, for constraints $C1..C3$ as well as the correctness of the synthesized program in the absence of faults, the same argument as given in Theorem 2 applies. Regarding satisfaction of timing constraints in the presence of faults, we observe that the property $(Q \mapsto_{\leq \delta} S)$ is satisfied based on the invocation of Add_BoundedResponse on Line 7. If the minimum delay from some state in Q to a state in S was greater than δ, then such states are removed. Hence, at the second invocation of Add_BoundedResponse, such states are not considered. As a result, adding other shortest paths to S does not increase the delay from states in Q. It follows that both timing constraints of graceful 2-phase recovery are satisfied.

Regarding completeness, the proof is similar to that of Theorem 2 as well. In particular, any state removed by Add_GracefulPhasedRecovery must be removed in any solution that meets the timing constraints of graceful 2-phase recovery. ∎

Next, we discuss the main differences between the two algorithms and identify the main reason that permits solution of graceful 2-phase recovery be in polynomial-time without closure of Q, but causes the addition of relaxed 2-phase recovery to be NP-complete. Observe that in Line 10 in Add_RelaxedPhasedRecovery, we added recovery paths from states in T_1 to states in Q. Without closure property of Q, the paths added for

Add_RelaxedPhasedRecovery can create cycles with paths added from Q to S. Such cycles outside S prevent the program from recovering to the invariant predicate within the required timing constraint. To the contrary, in Line 10 in Add_GracefulPhasedRecovery, we added recovery paths from states in T_1 to states in S. These paths cannot create cycles with paths added from $Q - S$. Moreover, the paths also do not increase the delay in recovering from Q to S. For this reason, the problem of Add_GracefulPhasedRecovery could be solved in polynomial-time.

5.5 Other Types of 2-Phase Recovery

Let us consider Definition 11 closely. Interesting possible values for Q_1 are Q, S and $Q - S$ and interesting possible values for Q_2 are $\neg S$ and Q. Thus, the different combinations for 2-phase recovery are as shown in the table below.

	$Q_1 = Q$	$Q_1 = S$	$Q_1 = Q - S$
$Q_2 = Q$	strict	graceful	ordered-strict
$Q_2 = \neg S$	relaxed	single phase	ordered-relaxed

Of these, we showed that the problem of relaxed 2-phase recovery is NP-complete. It is straightforward to observe that this proof can be extended to show that synthesizing ordered-relaxed 2-phase recovery is also NP-complete. Moreover, in [10], we showed that the problems of synthesizing strict and ordered-strict 2-phase recovery are NP-complete. As mentioned in Subsection 5.4, one surprising result in this table, however, is that the problem of synthesizing graceful 2-phase recovery can be solved in polynomial-time.

6 Case Study: Traffic Controller

We use a one-lane bridge traffic controller program to illustrate the application of algorithms in this paper. To concisely write the transitions of a program, we use *timed guarded commands.* A timed guarded command (also called *timed action*) is of the form $L :: g \xrightarrow{\lambda} st$, where L is a label, g is a state predicate, st is a statement that describes how the discrete variables are updated, and λ is a set of clock variables that are reset by execution of L. Thus, L denotes the set of transitions $\{(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0]) \mid g$ is true in state (s_0, ν), and s_1 is obtained by changing s_0 as prescribed by $st\}$. A *guarded wait command* (also called *delay action*) is of the form $L :: g \longrightarrow$ **wait**, where g identifies the set of states from where delay transitions with arbitrary durations are allowed to be taken as long as g continuously remains true.

Fault-intolerant program $\mathcal{TC}$. The one-lane bridge traffic controller program (denoted $\mathcal{TC}$) has two discrete variables sig_0 and sig_1 with domain $\{G, Y, R\}$. Moreover, for each signal i, $i \in \{0, 1\}$, $\mathcal{TC}$ has three clock variables x_i, y_i, and z_i acting as timers to change signal phase. When a signal turns green, it turns yellow within 1-10 time units. Subsequently, the signal may turn red between

1-2 time units after it turns yellow. Finally, when the signal is red, it may turn green within 1 time unit after the other signal becomes red. Both signals operate identically. The traffic controller program is as follows for $i \in \{0, 1\}$:

$$\begin{array}{lll}
\mathcal{TC}1_i :: & (sig_i = G) \wedge (1 \leq x_i \leq 10) & \xrightarrow{\{y_i\}} \quad (sig_i := Y); \\
[] & & \\
\mathcal{TC}2_i :: & (sig_i = Y) \wedge (1 \leq y_i \leq 2) & \xrightarrow{\{z_i\}} \quad (sig_i := R); \\
[] & & \\
\mathcal{TC}3_i :: & (sig_i = R) \wedge (z_j \leq 1) & \xrightarrow{\{x_i\}} \quad (sig_i := G); \\
[] & & \\
\mathcal{TC}4_i :: & ((sig_i = G) \wedge (x_i \leq 10)) \vee & \\
 & ((sig_i = Y) \wedge (y_i \leq 2)) \vee & \\
 & ((sig_i = R) \wedge (z_j \leq 1)) & \longrightarrow \quad \textbf{wait};
\end{array}$$

where $j = (i + 1) \mod 2$ and the operator [] denotes non-deterministic choice of execution. Notice that the guard of $\mathcal{TC}3_i$ depends on z timer of signal j. For simplicity, we assume that once a traffic light turns green, all cars from the opposite direction have already left the bridge.

Safety specification $SPEC_{\overline{bt}_{\mathcal{TC}}}$. The set of transitions that should not be included in the synthesized program, $SPEC_{bt_{\mathcal{TC}}}$, include transitions where no signal is red in the target states.

$$SPEC_{bt_{\mathcal{TC}}} = \{(\sigma_0, \sigma_1) \mid (sig_0(\sigma_1) \neq R) \wedge (sig_1(\sigma_1) \neq R)\}.$$

Invariant $S_{\mathcal{TC}}$. One invariant for the program $\mathcal{TC}$ is the following state predicate:

$$\begin{array}{llll}
S_{\mathcal{TC}} = \forall i \in \{0,1\} : & [(sig_i = G) & \Rightarrow & ((sig_j = R) \wedge (x_i \leq 10) \wedge (z_i > 1))] \wedge \\
 & [(sig_i = Y) & \Rightarrow & ((sig_j = R) \wedge (y_i \leq 2) \wedge (z_i > 1))] \wedge \\
 & [((sig_i = R) \wedge (sig_j = R)) & & \\
 & & \Rightarrow & ((z_i \leq 1) \oplus (z_j \leq 1))],
\end{array}$$

where $j = (i+1) \mod 2$ and $\oplus$ denotes the *exclusive or* operator. It is straightforward to see that $\mathcal{TC}$ satisfies $SPEC_{\overline{bt}_{\mathcal{TC}}}$ from $S_{\mathcal{TC}}$.

Faults. $\mathcal{TC}$ is subject to clock reset faults due to circuit malfunctions that reset z_0 and/or z_1. These actions are represented by the following guarded commands.

$$\begin{array}{lll}
F_0 :: & S_{\mathcal{TC}} & \xrightarrow{\{z_0\}} \quad \textbf{skip}; \\
F_1 :: & S_{\mathcal{TC}} & \xrightarrow{\{z_1\}} \quad \textbf{skip};
\end{array}$$

It is straightforward to see that in the presence of F_0 and F_1, $\mathcal{TC}$ may violate $SPEC_{\overline{bt}_{\mathcal{TC}}}$. For instance, if F_1 occurs when $\mathcal{TC}$ is in a state of $S_{\mathcal{TC}}$ where $(sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 > 1)$, in the resulting state, we have $(sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 = 0)$. From this state, immediate execution of timed actions $\mathcal{TC}3_0$ and then $\mathcal{TC}3_1$ results in a state where $(sig_0 = sig_1 = G)$, which is clearly a violation of the timing independent safety specification.

6.1 Application of Relaxed 2-Phase Recovery

Now, we use the algorithm Add_RelaxedPhasedRecovery for adding relaxed 2-phase recovery to obtain a fault-tolerant version of $\mathcal{TC}$.

Constraints for adding relaxed 2-phase recovery. We consider the following constraints for adding relaxed 2-phase recovery.

$$SPEC_{\overline{br_{\mathcal{TC}}}} \equiv (\neg S_{\mathcal{TC}} \mapsto_{\leq 3} Q_{\mathcal{TC}}) \ \wedge \ (\neg S_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}}),$$

where $Q_{\mathcal{TC}} = S_{\mathcal{TC}} \cup \forall i \in \{0,1\} \ : \ (sig_i = R) \ \wedge \ (z_i > 1)$. This timing constraint requires that if the program is perturbed to a state that is outside the invariant $S_{\mathcal{TC}}$, then within 7 time units, the program has to recover to $S_{\mathcal{TC}}$. Moreover, within 3 time units, the program must reach a state in $Q_{\mathcal{TC}}$, which includes states in $S_{\mathcal{TC}}$ and states where both signals are red and no signal can turn green using the original actions of $\mathcal{TC}$. Thus, the designer has the following tradeoffs: (1) the program can recover to $Q_{\mathcal{TC}} - S_{\mathcal{TC}}$ in γ time units ($\gamma \leq 3$) and then spend at most $7-\gamma$ time units to recover to $S_{\mathcal{TC}}$, or (2) the program can directly recover to $S_{\mathcal{TC}}$ in 3 time units.

To apply Add_RelaxedPhasedRecovery, first, in Step 1, we compute ms and mt. It is straightforward to observe that $ms = \{\}$ and $mt = SPEC_{bt_{\mathcal{TC}}}$. In Step 2, we add bounded response property $(Q_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}})$ (Line 8). This introduces a clock variable, say t_1 which is reset whenever a program executes a transition that begins in a region where $Q^r_{\mathcal{TC}}$ is false and ends in a region where $Q^r_{\mathcal{TC}}$ is true. Note that due to the closure requirement for $Q_{\mathcal{TC}}$, the program does not execute transitions that begin in a region where $Q^r_{\mathcal{TC}}$ is true and end in a region where $Q^r_{\mathcal{TC}}$ is false. Step 2 also adds the following recovery action:

$$\mathcal{TC}5_i \ :: \quad (sig_0 = sig_1 = R) \ \wedge \ (z_0, z_1 > 1) \quad \xrightarrow{z_i} \quad \textbf{skip};$$

for all $i \in \{0,1\}$. This action ensures that if the program is in a state in $Q_{\mathcal{TC}} - S_{\mathcal{TC}}$, then it can recover to a state in $S_{\mathcal{TC}}$. The delay involved in this transition is 0. Hence, for any region r in $Q^r_{\mathcal{TC}}$, $wt(r) = 0$. Note that Add_BoundedResponse can add additional delay transitions that do not violate the required timing constraints. However, as mentioned in Subsection 5.2, this addition must be done after ensuring the second timing constraint (Line 10).

Next, in Step 3, we apply the *transform* function and add $(T^r_1 \mapsto_{\leq \gamma} Q^r_{\mathcal{TC}})$. The value of γ depends on the region in $Q^r_{\mathcal{TC}}$ where recovery is added. Step 3 introduces a clock variable t_2 which is reset whenever a fault transition that begins in a region in $Q^r_{\mathcal{TC}}$ and ends in a region in $T^r_1 - Q^r_{\mathcal{TC}}$ occurs. Additionally, it adds recovery actions to ensure $(T^r_1 \mapsto_{\leq 3} Q^r_{\mathcal{TC}})$. To achieve this, Add_BoundedResponse adds the shortest path from every state in $T^r_1 - Q^r_{\mathcal{TC}}$ to a state in $Q^r_{\mathcal{TC}}$ as long as the delay of this shortest path is less than 3. It can also add other paths as long as the maximum delay for recovery to $Q^r_{\mathcal{TC}}$ does not exceed 3. Thus, some of the recovery action added for sig_0 are as follows:

$$\mathcal{TC}6_0 :: (sig_0 \neq R \vee sig_1 \neq R) \ \wedge \ (z_0, z_1 \leq 1) \ \wedge \ (t_2 \leq 1) \ \longrightarrow \quad (sig_0 := sig_0 := R);$$

This action forces that program to immediately turn both signals Red.

Tradeoff in relaxed 2-phase recovery. In the above analysis, we only considered the shortest paths from $Q^r_{\mathcal{TC}}$ to $S^r_{\mathcal{TC}}$. In adding relaxed 2-phase recovery, there is a tradeoff between the time used for recovering to $Q^r_{\mathcal{TC}}$ and the time used for recovering to $S^r_{\mathcal{TC}}$. For example, after the addition of shortest paths, we can add additional paths from $Q^r_{\mathcal{TC}}$ to $S^r_{\mathcal{TC}}$ as long as both the timing constraints are met. However, addition of some recovery paths from $Q^r_{\mathcal{TC}}$ to $S^r_{\mathcal{TC}}$ requires removal of some paths for recovery to $Q^r_{\mathcal{TC}}$. For example, we can add the following action for recovery from $Q^r_{\mathcal{TC}}$ to $S^r_{\mathcal{TC}}$.

$$\mathcal{TC}7_0 :: \quad (sig_0 = sig_1 = R) \wedge (t_1 \leq 5) \quad \longrightarrow \quad \textbf{wait};$$

However, if such an action is added then the weight of the state where both signals are red and $t_1 = 0$ is 5. Hence, recovery to such a state must be added in at most 2 time units.

6.2 Application of Graceful 2-Phase Recovery

To illustrate the application of Add_GracefulPhasedRecovery, we change the timing constraints to $SPEC_{\overline{br_{\mathcal{TC}}}} \equiv (\neg S_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}}) \wedge (Q_{\mathcal{TC}} \mapsto_{\leq 3} S_{\mathcal{TC}})$. These constraints require that if the program is perturbed to a state that is outside the invariant $S_{\mathcal{TC}}$, then within 7 time units, it recovers to $S_{\mathcal{TC}}$. Moreover, if the fault only perturbs the program to $Q_{\mathcal{TC}}$, then recovery must complete within 3 time units.

The first step of Add_GracefulPhasedRecovery is the same as that of Add_RelaxedPhasedRecovery. In the second step (Line 7), we compute transitions that can be used in adding fault-tolerance. Since $Q_{\mathcal{TC}}$ need not be closed for Add_GracefulPhasedRecovery, the transitions computed in Line 7 contain additional transitions where the program begins in a region where $Q^r_{\mathcal{TC}}$ is true and ends in a region where $Q^r_{\mathcal{TC}}$ is false. Examples of such transitions include transitions that turn at most one signal to green or yellow.

Subsequently, we add $(Q_{\mathcal{TC}} \mapsto_{\leq 3} S_{\mathcal{TC}})$. This addition introduces a clock variable, say t_1, which is reset whenever a program executes a transition that begins in a region where $Q^r_{\mathcal{TC}}$ is false and ends in a region where $Q^r_{\mathcal{TC}}$ is true. In addition, it adds shortest recovery paths from each state in $Q_{\mathcal{TC}}$ to a state in $S_{\mathcal{TC}}$. It turns out that the shortest paths from from $Q_{\mathcal{TC}}$ is not affected by the new transitions included on Line 7. Hence, the program adds the same action, $\mathcal{TC}5_i$ as in Add_RelaxedPhasedRecovery.

Then, the program adds $(\neg S_{\mathcal{TC}} \mapsto_{\leq 7} S_{\mathcal{TC}})$. To achieve this, Add_BoundedResponse adds the shortest path from every region in $T^r_1 - S^r_{\mathcal{TC}}$ to a region in $Q^r_{\mathcal{TC}}$ as long as the length of this path is less than 7. Note that if $Q_{\mathcal{TC}}$ had any state from where the minimum delay was more than 3, then such a state would have been removed while ensuring the first bounded response property. Hence, the delay for the shortest paths from $Q_{\mathcal{TC}}$ would be at most 3. Subsequently, it can also add other paths as long as the maximum delay for recovery to $Q^r_{\mathcal{TC}}$ does not exceed 3 and the maximum delay from any state in $\neg S_{\mathcal{TC}}$ would not exceed 7.

7 Conclusion

In this paper, we focused on complexity analysis of synthesizing bounded-time 2-phase recovery. This type of recovery consists of two bounded response properties of the form: $(\neg S \mapsto_{\leq \theta} Q_1) \ \wedge \ (Q_2 \mapsto_{\leq \delta} S)$. We characterized S as an ideal behavior and $Q_{1,2}$ as acceptable intermediate behaviors during recovery. Each property expresses one phase of recovery within the respective time bounds θ and δ in a fault-tolerant real-time program. We formally defined different scenarios of 2-phased recovery, characterized their applications in real-world systems, and considered two types of them called relaxed (where $Q_1 = Q$ and $Q_2 = \neg S$) and graceful (where $Q_1 = S$ and $Q_2 = Q$). We showed that, in general, the problem of synthesizing relaxed 2-phase recovery is NP-complete. However, the problem can be solved in polynomial-time, if $S \subseteq Q$ and Q is closed in the synthesized program. We also found a surprising result that the problem of synthesizing graceful 2-phase recovery can be solved in polynomial-time even though all other variations are NP-complete. We emphasize that all complexities are in the size of the input program's region graph.

Based on the complexity analysis, we find that the problem of synthesizing relaxed 2-phase recovery is significantly simpler, if the intermediate recovery predicate Q is closed in the execution of the synthesized program. This result implies that if the intermediate recovery predicate is used for *recording* the fault, then the complexity of the corresponding problem is substantially higher than the case where the program quickly provides acceptable behavior.

One future research direction is to develop heuristics to cope with the NP-complete instances. Based on our experience with synthesizing distributed fault-tolerant programs [9, 11], we believe that efficient implementation of such heuristics makes it possible to synthesize real programs in practice. Another research problem is to consider the case where a real-time program is subject to different classes of faults and a different type of tolerance is required for each fault class.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1-2):86–109, 1997.
4. E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control (HSCC)*, pages 19–30, 1999.
5. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.
6. J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications.* Springer, 2002.

7. B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS 4346, pages 261–276, 2006.
8. B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4280, pages 122–136, 2006.
9. B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.
10. B. Bonakdarpour and S. S. Kulkarni. Masking faults while providing bounded-time phased recovery. In *International Symposium on Formal Methods (FM)*, pages 374–389, 2008.
11. B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*, pages 167–171, 2008.
12. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.
13. L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *International Conference on Concurrency Theory (CONCUR)*, 2003.
14. D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 571–582, 2002.
15. M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.
16. S. Fortune, J. E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.
17. T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.