Noname manuscript No.
(will be inserted by the editor)

Synthesizing Bounded-Time 2-Phase Recovery

Borzoo Bonakdarpour · Sandeep S. Kulkarni

Abstract We focus on synthesis techniques for transforming existing fault-intolerant real-time programs into fault-tolerant programs that provide *phased recovery*. A fault-tolerant program is one that satisfies its *safety* and *liveness* specifications as well as *timing constraints* in the presence of faults. We argue that in many commonly considered programs (especially in mission-critical systems), when faults occur, simple recovery to the program's normal behavior is necessary, but not sufficient. For such programs, it is necessary that recovery is accomplished in a sequence of phases, each ensuring that the program satisfies certain properties. In the simplest case, in the first phase the program recovers to an *acceptable* behavior within some time θ, and, in the second phase, it recovers to *ideal* behavior within time δ. In this article, we introduce four different types of bounded-time 2-phase recovery, namely ordered-strict, strict, relaxed, and graceful, based on how a real-time fault-tolerant program reaches the acceptable and ideal behaviors in the presence of faults. We rigorously analyze the complexity of automated synthesis of each type: we either show that the problem is hard in some class of complexity or we present a sound and complete synthesis algorithm. We argue that such complexity analysis is essential to deal with the highly complex decision procedures of program synthesis.

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

A preliminary version of this manuscript appeared in International Conference on Formal Methods (FM'08 [13] and FM'09 [16]).

B. Bonakdarpour
School of Computer Science, University of Waterloo, 200 University Avenue West, N2L 3G1, Waterloo, Ontario, Canada
E-mail: borzoo@cs.uwaterloo.ca

S. S. Kulkarni
3115 Engineering Building, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA
E-mail: sandeep@cse.msu.edu

1 Introduction

Achieving correctness is perhaps the most important reason to apply formal methods in design and development of computing systems. Such correctness 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. Thus, *time-predictability* and *fault-tolerance* are two essential properties of programs that operate in such systems. Nevertheless, reasoning about the correctness of these properties has always been a challenge, as they have conflicting natures for the following reasons:

- To guarantee fault-tolerance, one has to deal with the occurrence of *unpredictable* faults and delays, so that the program does not violate its safety and eventually recovers to its legitimate states from where its subsequent behavior is *ideal* (i.e., one that occurs in the absence of faults).
- Meeting real-time constraints requires a program to act in a deterministic and predictable fashion.

One way to deal with this challenge is to design automated synthesis algorithms that build fault-tolerant real-time 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., [18,22,6,7]) and *game theory* (e.g., [20,23]). In controller synthesis (respectively, game theory), program and *fault* transitions can be modeled as controllable and uncontrollable actions (respectively, in terms of two players) and the objective is to constrain the actions of a *plant* (respectively, a *player*) at each state through synthesizing a *controller* (respectively, a *wining strategy*). The synthesized constraints ensure that the entire system always meets its safety and/or reachability properties. However, the notion of fault-tolerance requires other features that are not typically addressed in controller synthesis and game theory. *Bounded-time recovery*, where a program returns to its normal behavior when its state is perturbed by the occurrence of faults, is one such feature. In this context, a fault-intolerant program normally lacks this feature and a recovery mechanism must be *added* to the program, so that it reacts to faults properly.

In many commonly considered systems, achieving recovery within a certain time-bound to the ideal behavior in the presence of faults is necessary, but not sufficient and sometimes not feasible. Our idea to deal with this problem is to ensure that the program recovers quickly to an *acceptable* behavior and eventually recovers to its *ideal* behavior. An acceptable behavior is one that the system operates in a degraded fashion where the system cannot exhibit its full functionality for a bounded amount of time. Such a constraint ensures that the system first goes to a state in which a set of preconditions for final recovery (e.g., via a system reboot or rollback) is fulfilled. To better motivate this idea, we present the following example.

1.1 Motivating Example

Consider a one-lane turn-based bridge where cars can travel in only one direction at any time. The bridge is controlled by two traffic signals, say sig_0 and sig_1, at the two ends of the bridge. The signals work as follows. Each signal changes phase from green to yellow and then to red, based on a set of timing constraints. Moreover, if one signal is red, it will turn green some time after the other signal turns red. Thus, at any time, the values of sig_0 and sig_1 show in which direction cars are travelling. The *specification* of this system can be easily characterized by a set $SPEC_{bt}$ of bad transitions that reach states where both signals are not red at the same time. In order to address the correctness of the system, we identify a system *invariant*: a set S of states from where the system behaves correctly. In fact, S characterizes the *ideal* behavior of the system. In case of the traffic signals system, one system invariant is the set of states from where the system always reaches states where at least one signal is red and they change phases in time. As long as the system's state is in S, nothing catastrophic will happen. However, this is not the case when the system is subject to a set of faults.

Let us consider a scenario where the state of the systems is perturbed by occurrence of a fault that causes the system to reach a state, say s, in $\neg S$. Although reaching s may not necessarily violate the system specification, subsequent signal operations can potentially result in execution of a transition in $SPEC_{bt}$. For example, when sig_0 is green and sig_1 is red, if the responsible timer for changing sig_1 from red to green gets reset due to a circuit problem, sig_1 may turn green within some time while sig_0 is also green. Thus, our system is fault-intolerant, as it violates its specification in the presence of faults.

In order to transform this system into a fault-tolerant one, it is desirable to synthesize a version of the original system, in which even in the presence of faults, the system (1) never executes a transition in $SPEC_{bt}$, and (2) always meets the following bounded-time recovery specification denoted by $SPEC_{br}$: When the system state is in $\neg S$, it must reach a state in S within a bounded amount of time. Although such a recovery mechanism is necessary in a fault-tolerant real-time system, it may not be feasible or in some cases sufficient. In particular, one may require a 2-phase recovery mechanism where the system must initially reach a special set of *acceptable* states, say Q, within some time θ, and subsequently recovers to S within δ time units. In our example, one possibility for Q, say Q_1, could be the set of states where all signals remain red for a long enough time (e.g., in order to ensure that nothing disastrous happens). Another possibility for Q, say Q_2, could include the set of all normal states of the signal and additional states where the signal remains red for a long enough time. One of the main results in the paper is that for predicates such as Q_1 —where it is expected that the system will reach a state in Q_1 and subsequently leave Q_1 to reach a legitimate state— the complexity of adding 2-phase recovery is high (NP-complete). By contrast, for predicates such as Q_2 —where the legitimate states are a subset of Q_2 and it is expected that the revised program never goes from a state in Q_2 to a state in $\neg Q_2$— the complexity of adding 2-phase recovery is low (polynomial-time).

1.2 Instances of 2-phase Recovery in Practice

The application of 2-phased recovery can also be found in design and/or verification of several protocols. For example, in [28], Gouda and Multari have utilized multi-phase recovery for design of stabilizing window protocol (similar to that used in TCP) as well as stabilizing handshake protocol. Some of the differences between our work and their work are as follows: They provide an instance of protocol that provide multi-phase recovery whereas we focus on automated synthesis of protocols that provide 2-phase recovery. Specifically, the stabilizing window protocol in [28] is an instance of

3-phase recovery whereas the stabilizing handshake is an instance of 4-phase recovery. Another difference is that they focus on untimed versions of multi-phased recovery (i.e., $\delta = \theta = \infty$). By contrast, our algorithms are applicable even if timing constraints are finite. Moreover, they assume that the intermediate predicates (e.g., predicate Q in above discussion) are closed. Based on the results in Section 6.2, we observe that such problems can be solved in polynomial-time. Also, our algorithm can be easily extended for multi-phased recovery. (Remark 5 in Section 6.1 provides more details about this claim.).

In [4], Arora presents a tree construction algorithm in a distributed system (e.g., for mutual exclusion) that provides 2-phase recovery. Specifically, in this protocol, in the first phase, it is guaranteed that after faults stop occurring, there will be no unrooted trees. This ensures that every node will have a path to the root of the tree it is involved in. The second phase is responsible for ensuring that a unique tree is formed. Thus, [4] is an instance of 2-phase recovery where the intermediate predicate corresponds to 'no unrooted trees'. The timing bounds for such recovery are identified in terms of *steps* instead of *time*. Other protocols that provide multi-phase recovery include [25,27,21].

Finally, consider a system that controls the pressure of a boiler. The boiler should normally operate within 10-20 atmospheres. It can also operate safely in 20-40 atmospheres, but only for a short time. Faults can suddenly change the pressure to over 40 atmospheres. Recovery should be carried out, so that the boiler does not experience sudden reduction of pressure. To this end, two valves are provided to reduce the pressure. One valve responds quickly, but reduces the pressure slowly. A second valve reacts slowly, but reduces the pressure quickly. Thus, the first phase of recovery involves opening the first valve to reduce the pressure as soon as possible even with a small volume. When the system reaches an "acceptable" level of 20-40, during the second phase, the second valve gets open to further reduce the pressure to 10-20. In fact, any recovery mechanism that has to be carried out in multiple steps where each step ensures a set of constraints can be modeled by our notion of 2-phase recovery.

1.3 Contributions

We model a recovery constraint by a *bounded-time response* property. This property is of the form:

$$(A \mapsto_{\leq \alpha} B),$$

i.e., starting from any state in A, the program reaches a state in B within α time units. Following the above example, we identify three recovery constraints: (1) $(\neg S \mapsto_{\leq \delta} S)$, (2) $(\neg S \mapsto_{\leq \theta} Q)$, and (3) $(Q \mapsto_{\leq \theta} S)$. We distinguish different variants for such 2-phase recovery problem based on a combination of these constraints:

- The scenario discussed above can be expressed in terms of constraints of the form:

  $$(\neg S \mapsto_{\leq \theta} Q) \;\wedge\; (Q \mapsto_{\leq \delta} S),$$

  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:

$$(\neg S \mapsto_{\leq\theta} (Q - S)) \;\wedge\; (Q \mapsto_{\leq\delta} S),$$

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:

$$(\neg S \mapsto_{\leq\theta} Q) \;\wedge\; (\neg S \mapsto_{\leq\delta} S),$$

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 a requirement 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

$$(Q \mapsto_{\leq\delta} S) \;\wedge\; (\neg S \mapsto_{\leq\theta} S),$$

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.

We analyze the complexity of synthesizing these four variations of 2-phase recovery. Although synthesis algorithms are generally known to be intractable, it has been shown that their complexity can be overcome through rigorous complexity analysis of specific classes of properties. Such analysis identifies bottlenecks of synthesis which leads to devising intelligent heuristics. Previously, we have shown that by efficient implementation of these heuristics, we can effectively synthesize large fault-tolerant distributed programs [12,15], where the corresponding synthesis problem is NP-complete in the size of state space of the intolerant program.

The main contributions of the paper are as follows (Appendix A presents a summary of results in a graphical fashion):

- We formally define and classify different types of bounded-time 2-phase recovery in the context of fault-tolerant real-time programs.
- Regarding synthesizing strict and relaxed 2-phase recovery, we show that
  - The general problems are NP-complete.
  - The problems remain NP-complete even if $S \subseteq Q$ and $\delta = \infty$.
  - The problems remain NP-complete even if $\theta = \infty$ and $\delta = \infty$.
  - The problems can be solved in polynomial-time if $S \subseteq Q$ and $\theta = \infty$.
  - the problem can be solved in polynomial-time, if it is required that Q must be closed, i.e., the synthesized program cannot begin in a state in Q and reach a state outside Q.

– Regarding synthesizing ordered-strict 2-phase recovery, we show that the problem remains NP-complete even if $S \subseteq Q$, $\theta = \infty$, and $\delta = \infty$.
– 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* [2] (i.e., a time-abstract finite bisimulation of a real-time program). In the worst case, the size of the region graph can be exponential in the size of the timed automata. A sketch for the algorithm for constructing the region graph (from [2]) is presented in Section 6.

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 bounded-time 2-phase recovery. In Section 4, we define the problem statement for synthesizing 2-phase recovery. In Section 5, we show that in general, the complexity of synthesis of strict, ordered-strict, and relaxed 2-phase recovery is NP-complete. We also consider a set of subproblems that remain NP-complete and those that can be solved in polynomial-time. Our more sophisticated sufficient conditions on synthesizing strict and relaxed 2-phase recovery in polynomial-time are presented in Section 6. Graceful 2-phase recovery is studied in Section 7. Related work is discussed in Section 8. Finally, we make concluding remarks and discuss future work in Section 9. Appendix A presents a summary of results in a graphical fashion.

2 Preliminaries

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 [29].

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 valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. A *clock constraint* over the set X of clock variables is a Boolean combination of formulas 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 $\Phi(X)$.

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:* $(s_0, \nu) \to (s_1, \nu[\lambda := 0])$, 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:* $(s, \nu) \to (s, \nu + \delta)$, 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 from 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

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

2.1.1 Example

We use a one-lane bridge traffic controller program to describe the concepts and 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 $\{(s_0, \nu) \to (s_1, \nu[\lambda := 0]) \mid g$ 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.

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 $i \in \{0, 1\}$, $\mathcal{TC}$ 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 and 10 time units. Subsequently, the signal may turn red between 1 and 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 $i \in \{0, 1\}$:

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

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.

2.2 Specification

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 $\tau_i = \tau_{i+1}$. ■

The *consistency* constraint requires that all clock variables advance with the same rate in conformance to a global clock. 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*.

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 [29], 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).

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 [1,29]. 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. ∎

2.3 Example (cont'd)

Continuing our traffic controller example, the set $SPEC_{bt_{\mathcal{TC}}}$ of immediate bad transitions 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:

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

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 $S_{\mathcal{TC}}$.

3 Fault Model and Fault-Tolerance

Our fault model and the notion of fault-tolerance is adapted from the work by Arora and Gouda [5] extended to the context real-time systems in [11].

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.

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

Table 1 The four types of 2-phase recovery.

3.1.1 Example (cont'd)

$\mathcal{TC}$ 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.

$$F_0 :: \; S_{\mathcal{TC}} \quad \xrightarrow{\{z_0\}} \quad \textbf{skip};$$
$$F_1 :: \; S_{\mathcal{TC}} \quad \xrightarrow{\{z_1\}} \quad \textbf{skip};$$

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.

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 shown in Table 1. We call θ and δ *intermediate recovery time* and *recovery time*, respectively. ∎

We are now ready to define what it means for a program to be *fault-tolerant*. Intuitively, a fault-tolerant program satisfies its safety, liveness, and timing constraints in both absence and presence of faults. In other words, the program *masks* the occurrence of faults in the sense that all program requirements are persistently met in the absence and presence of faults[1].

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

[1] In [11], we introduced different levels of fault-tolerance for real-time programs. Our notion of fault-tolerance in this paper is equivalent to the *hard-masking* level of fault-tolerance as defined in [11]. Hard-masking is strongest level of fault-tolerance and weaker levels are possible by omitting the requirement of satisfying safety, liveness, timing constraints, or a combination of them in the presence of faults.

8 and 9. We say that $\mathcal{P}$ is *f-tolerant to SPEC from S*, iff (1) $\mathcal{P} \models_S SPEC$, (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, and (3) $\langle S_\mathcal{P}, \psi_\mathcal{P}[]f\rangle$ satisfies $T \mapsto_{\leq\infty} S$ 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".

3.2.1 Example (cont'd)

When faults F_0 or F_1 (defined in Subsection 3.1) occur, the program $\mathcal{TC}$ has to ensure that nothing catastrophic happens and also recover to its normal behavior. Thus, we would like the fault-tolerant version of $\mathcal{TC}$ to reach a state where both signals remain red for a long enough time or where regular operation is resumed. In particular, we let the (for example) strict 2-phase recovery specification of $\mathcal{TC}$ be the following:

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

where $Q_{\mathcal{TC}} = \forall i \in \{0,1\} \;:\; (sig_i = R) \;\wedge\; (z_i > 1)$. The response times in $SPEC_{\overline{br}_{\mathcal{TC}}}$ are just two arbitrary numbers to express the duration of the two phases of recovery.

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 $\psi^s_\mathcal{P}$ that start in S and end in S, and delay transitions of $\psi^d_\mathcal{P}$ 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:

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

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 $S' \subseteq S$. 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 $\psi_{\mathcal{P}'}|S' \subseteq \psi_\mathcal{P}|S'$. Finally, we require that the given fault-intolerant program has built-in clock variable for each bounded response property involved in 2-phase recovery specification $SPEC_{\overline{br}}$. This assumption is needed by synthesis algorithms to measure the time elapsed since respective state predicates has become true.

Assumption 1 Let $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P}\rangle$ be a real-time program and let $(A \mapsto_{\leq\alpha} B)$, $\alpha \neq \infty$ be a recovery property. If $\mathcal{P}$ satisfies $(A \mapsto_{\leq\alpha} B)$ or if the recovery property $(A \mapsto_{\leq\alpha} B)$ is to be added to $\mathcal{P}$ by a synthesis algorithm, we assume that $\mathcal{P}$ has a clock variable that gets reset whenever $(A \wedge (\neg B))$ becomes true.

Remark 2 If $\mathcal{P}$ is to be revised to add several properties of the form $(A_1 \mapsto_{\leq \alpha_1} B_1)$, $(A_2 \mapsto_{\leq \alpha_2} B_2)$, ..., then the clock variable that gets reset for these properties may be (but is not required to be) the same. Also, the variable that gets reset for $(A \mapsto_{\leq \alpha} B)$ could be reset in other scenarios. However, it must get reset when the program state is changed from a state in $\neg((A \wedge (\neg B)))$ to a state in $(A \wedge (\neg B))$. Note that these transitions essentially 'start' the clock on the corresponding leads-to property. If the timing constraints are ∞ then such a clock variable is not needed since we do not need to compute precise delays. The reason for this assumption is the algorithm in [10] to compute delays assumes the existence of such a clock variable.

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 $\mathcal{P} \models_S SPEC$ and satisfies Assumption 1, identify $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ and S' such that:

- $(C1)$ $S' \subseteq S$,
- $(C2)$ $\psi_{\mathcal{P}'} \mid S' \subseteq \psi_{\mathcal{P}} | S'$, and
- $(C3)$ $\mathcal{P}'$ is f-tolerant to $SPEC$ from S'. ∎

The above problem statement can be instantiated for all four types of 2-phase recovery. Notice that conditions $C1$ and $C2$ in Problem Statement 1 precisely express the notion of behavior restriction in the sense of *language inclusion* used in controller synthesis and game theory. Moreover, constraint $C3$ 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 $C3$ allows the output program to exhibit recovery computations that input program does not have.

5 Complexity Analysis for Ordered-Strict, Strict and Relaxed 2-Phase Recovery

In this section, first, in Theorem 1, 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. Subsequently, in Corollary 1, we utilize this result to show that the problem remains NP-complete even if $S \subseteq Q$ and $\delta = \infty$. Then, in Theorem 2, we show that the problem of synthesizing fault-tolerant programs that provide strict 2-phase recovery is NP-complete even if $S \subseteq Q$ and $\delta = \infty$. Afterwards, in Theorems 3 and 4, we show that the problem of synthesizing fault-tolerant real-time programs that provide strict or relaxed 2-phase recovery can be solved in polynomial time if $S \subseteq Q$ and $\theta = \infty$. In Theorem 5, we show that the problem remains NP-complete even if $\theta = \infty$ and $\delta = \infty$. This implies that the problem of synthesizing programs that provide ordered-strict 2-phase recovery is NP-complete even if $S \subseteq Q$, $\theta = \infty$ and $\delta = \infty$ (cf. Theorem 6).

To show our first result relating the complexity of relaxed 2-phase recovery, we describe an instance of this problem, next. We also describe the simplified 2-path problem that we use to demonstrate NP-completeness of relaxed 2-phase recovery.

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* [24] 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 [8].

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*) $\psi_{\mathcal{P}} =$
$$\{(s_u, x = 1) \rightarrow (s_v, x := 0) \mid (u, v) \in A \ \wedge \ u \neq v_3\} \ \cup$$
$$\{(s_{v_3}, x = |A| + 1) \rightarrow (s_l, x := 0)\} \ \cup$$
$$\{(s_l, x = 1) \rightarrow (s_l, x := 0)\}.$$
- (*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 $\theta = |A|$.

An intuitive description of the above mapping is as follows. First, we include one clock variable x in $\mathcal{P}$. (Observe that the role of variable x in above transitions satisfies Assumption 1.) 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 all locations. The program invariant S merely includes states where the location is s_l. The set of program transitions consists of:

1. all arcs in A except arcs that originate at v_3,
2. a transition from s_{v_3} to s_l, and
3. 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 that correspond to the arcs of the original graph is 1. And, the delay on the transition from s_{v_3} to s_l is $|A| + 1$. 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

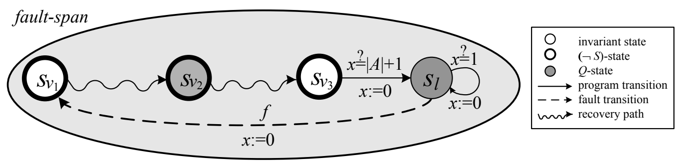

Fig. 1 Mapping the 2-path problem to synthesizing relaxed 2-phase recovery.

set of all transitions except those identified above. Thus, the program cannot use other (new) transitions to satisfy the recovery constraints, 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 $(s_{v_3}, x = |A| + 1) \rightarrow (s_l, x := 0)$, 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) $S' = S$, (2) $\psi_{\mathcal{P}'}|S' \subseteq \psi_{\mathcal{P}}|S'$, and (3) $\mathcal{P}'$ is fault-tolerant to $SPEC$ from S'. The latter holds because:
  (i) In the absence of faults, by starting from the invariant S', all computations of $\psi_{\mathcal{P}'}$ are infinite.
  (ii) In the presence of faults, $\mathcal{P}' \models_{S'} SPEC_{\overline{br}}$, since $\mathcal{P}' \models_{S'} SPEC_{\overline{bt}}$, and, Π is a simple path that reaches Q and S in the desired timing constraints.
- ($\Leftarrow$) 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' and, hence, location 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}. Observe that in $\overline{\sigma}_p$ a state cannot be visited more than once; i.e., the prefix does not include cycles. If this is not the case

then there exists a computation of the synthesized program that never reaches s_{v_3}. Furthermore, consider any transition in $\overline{\sigma}_p$ that changes location from s_{v_i} to s_{v_j}. Based on the definition of above safety specification, this transition must reset x. Hence, if any location, say s_{v_i} is visited more than once separately (i.e., ignoring the situation where the location remains unchanged due to delay transitions) then the state $(s_{v_i}, x = 0)$ is visited more than once. (Note that since new clock variables are not added, this region cannot be further subdivided.) Since the above discussion prevents repetition of such a state, it follows that a location cannot be repeated more than once in $\overline{\sigma}_p$. Thus, the path, say Π, whose vertices and arcs correspond to state and transitions in $\overline{\sigma}_p$, is a simple path and 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 $S \subseteq Q$ and $\delta = \infty$.

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 $S \subseteq Q$ and $\delta = \infty$.* ∎

Remark 3 Note that the above corollary implies that the problem is NP-complete for the case where it is given that $S \subseteq Q$ and $\delta = \infty$ as well as for the case where this condition is not specified. In particular, the discussion above the corollary shows that the problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides relaxed 2-phased recovery is NP-complete if $S \subseteq Q$ and $\delta = \infty$. Moreover, from Theorem 1, it is clear that the problem of transforming a fault-intolerant program into a fault-tolerant program that provides relaxed 2-phased recovery is NP-complete in the more general case.

Next, we show that the problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides strict 2-phase recovery is also NP-complete. While a proof similar to that of Theorem 1 can be obtained for this, we utilize Corollary 1 to obtain this result more easily. In particular, Corollary 1 shows that the problem providing relaxed 2-phase recovery is NP-complete even if $\delta = \infty$. And, if $\delta = \infty$ then we can show that it is possible to reduce the problem of providing relaxed 2-phased recovery to the problem of adding strict 2-phased recovery. To show this, we make the following observations.

Observation 1

$$(\neg S \mapsto_{\leq \theta} Q) \ \wedge \ (\neg S \mapsto_{\leq \infty} S)$$
$$\Rightarrow$$
$$(\neg S \mapsto_{\leq \theta} Q) \ \wedge \ (Q \mapsto_{\leq \infty} S)$$

Proof This result follows from the fact that if $(\neg S \mapsto_{\leq \infty} S)$ then $(Q \mapsto_{\leq \infty} S)$. ∎

Observation 2

$$(\neg S \mapsto_{\leq \theta} Q) \ \wedge \ (Q \mapsto_{\leq \infty} S)$$
$$\Rightarrow$$
$$(\neg S \mapsto_{\leq \theta} Q) \ \wedge \ (\neg S \mapsto_{\leq \infty} S)$$

Proof This result follows from the fact that the unbounded response relation is transitive, i.e., if $(\neg S \mapsto_{\leq \theta} Q)$ and $(Q \mapsto_{\leq \infty} S)$, then $(\neg S \mapsto_{\leq \infty} S)$. ∎

Based on these two observations, if $\delta = \infty$ then a solution for relaxed 2-phased recovery can be used to solve the strict 2-phased recovery and vice versa.

Theorem 2 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* strict *2-phased recovery is NP-complete in the size of locations of the fault-intolerant program even if $S \subseteq Q$ and $\delta = \infty$*

Proof This proof follows from Observations 1, 2 and Corollary 1. ∎

Finally, since a specialized instance of strict 2-phased recovery is NP-complete, the general problem is at least NP-hard. Moreover, since membership in NP is straightforward, we have:

Corollary 2 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* strict *2-phase recovery is NP-complete in the size of locations of the fault-intolerant program.*

Since the problem of relaxed and strict 2-phased recovery is NP-complete even if $\delta = \infty$, a natural question is 'What happens if $\theta = \infty$'. Next, we show that if $S \subseteq Q$ and $\theta = \infty$, then the problem of synthesizing relaxed and strict 2-phased recovery can be solved in polynomial-time in the size of locations of the given intolerant program.

Theorem 3 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* relaxed *2-phased recovery can be solved in polynomial time in the size of locations of the fault-intolerant program if $S \subseteq Q$ and $\theta = \infty$.*

Proof If $S \subseteq Q$, then $(\neg S \mapsto_{\leq \delta} S) \quad \Rightarrow \quad (\neg S \mapsto_{\leq \infty} Q)$. It follows that our problem of providing relaxed 2-phased recovery can be reduced to adding $(\neg S \mapsto_{\leq \delta} S)$. Since adding one such bounded response property can be achieved in polynomial-time in the size of the locations of the given program [10], the theorem holds[2].

Theorem 4 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* strict *2-phased recovery can be solved in polynomial time in the size of locations of the fault-intolerant program even if $S \subseteq Q \wedge \theta = \infty$*

Proof Since synthesizing this type of strict 2-phase recovery can be reduced to synthesizing graceful 2-phase recovery, we prove this theorem as a corollary of Theorem 14 in Section 7 that identifies the complexity of graceful 2-phased recovery. We note that this theorem will not be used to prove any other theorem until Theorem 14 to avoid circular reasoning. ∎

Since the problem of synthesizing strict and relaxed 2-phased recovery can be solved in polynomial-time if $S \subseteq Q$ and $\theta = \infty$, the natural question is whether the complexity of the problem changes if only one of the two conditions is satisfied. From Corollary 1, it follows that if only $S \subseteq Q$ is satisfied then the problem is NP-complete. For the case where $\theta = \infty$, we show the following.

Theorem 5 *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 $\theta = \infty$ and $\delta = \infty$.*

[2] In [10], we presented the algorithm Add_BoundedResponse for adding a single bounded response property to a given real-time program. We will provide an outline of this algorithm in Section 6.

Proof Once again, the membership in NP is straightforward. Hence, we only show that the problem is NP-hard. The proof is similar to that of Theorem 1. However, certain changes have to be made due to the fact that θ needs to be ∞. We denote the problem of relaxed 2-phased recovery with $\theta = \infty$ and $\delta = \infty$ as R2P$_{untimed}$. We reduce the 2PP problem to R2P$_{untimed}$.

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$_{untimed}$ problem (i.e., $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$, S, f, and $SPEC$) as follows (the main differences with that of Proof of Theorem 1 are listed in sens-serif):

- (clock variables) $X = \{\}$,
- (*state space*) $S_\mathcal{P} = \{s_v \mid v \in V\} \cup \{s_l\}$.
- (*program transitions*) $\psi_\mathcal{P} = (\{(s_u, s_v) \mid (u, v) \in A\} \cup \{(s_{v_3}, s_l) \cup \{(s_l, s_l)\} - \{(s_{v_3}, s_u) \mid u \in V - \{v_3\}\}$,
- (*invariant*) $S = \{s_l\}$.
- (*fault transitions*) $f = \{(s_l, s_{v_1}\}$.
- (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 $Q = \{s_{v_2}\}$, $\delta = \infty$, and $\theta = \infty$.

Thus, the main changes are as follows. Since there is no finite bound on the time to reach Q or S, there is no need for the clock variable. Hence, state space, program and fault transitions and invariant are changed to reflect that. Additionally, Q is changed to only include state s_{v_2}.

Reduction. Given the above mapping, we now show that 2PP has a solution iff the answer to R2P$_{untimed}$ problem is affirmative.

- ($\Rightarrow$) Let the answer to the 2PP be a simple path Π that originates at v_1, ends at v_3, and contains v_2. For this case, the corresponding argument from Theorem 1 shows that the answer to R2P$_{untimed}$ problem is affirmative.
- ($\Leftarrow$) Let the answer to the R2P$_{untimed}$ problem be $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ with invariant S'. Since S' must be nonempty, $S' = \{s_l\}$. 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 $\mathcal{P}'$. Based on the constraints of R2P$_{untimed}$, $\mathcal{P}'$ must reach s_{v_2} and s_l. Moreover, if $\mathcal{P}'$ reaches s_l first, then it cannot reach s_{v_2} since there is only a self-loop transition at s_l. Hence, $\mathcal{P}'$ must first reach s_{v_2}. Now, consider the computation of $\mathcal{P}'$ that begins from s_{v_1} and reaches s_l. Clearly, it reaches s_{v_2} first and then s_{v_3}. Also, this sequence cannot have a repeated state, as that would violate the guarantee that $\mathcal{P}'$ reaches S'. Hence, the path obtained from this computation is a simple path, i.e., the answer to the corresponding 2PP problem is affirmative. ∎

Since the result in Theorem 5 applies for the case where $\delta = \infty$, based on Observations 1 and 2, we have:

Corollary 3 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* strict *2-phased recovery is NP-complete in the size of locations of the fault-intolerant program even if $\theta = \infty$ and $\delta = \infty$.*

For the case where one desires ordered-strict 2-phased recovery, it is required that the program reaches $Q - S$ before it reaches S. Hence, from Corollary 3, we have

Theorem 6 *The problem of transforming a fault-intolerant real-time program into a fault-tolerant program that provides* ordered-strict *2-phased recovery is NP-complete in the size of locations of the fault-intolerant program even if $S \subseteq Q$, $\theta = \infty$ and $\delta = \infty$.*

To summarize the results from this section, we considered the complexity of strict, relaxed and ordered-strict 2-phased recovery. We showed that the general problem is NP-complete. For possible polynomial-time solutions, we considered three factors: $S \subseteq Q$, $\theta = \infty$ and $\delta = \infty$. For ordered-strict 2-phased recovery, the problem remains NP-complete even if all three conditions are satisfied. For relaxed and strict, the problem can be solved in polynomial-time if the first two constraints ($S \subseteq Q$, $\theta = \infty$) are satisfied. However, if any other combination of two constraints is satisfied, the problem remains NP-complete. In Section 6, we consider additional constraints under which the problem of synthesizing strict and relaxed 2-phased recovery can be solved in polynomial-time.

6 Polynomial-Time Solution for Strict and Relaxed 2-Phase Recovery with Closure of Q

In this section, we present another sufficient condition under which one can devise a polynomial-time sound and complete solution to the problem of transforming a fault-intolerant real-time program into a fault-tolerant program that satisfies strict (respectively, relaxed) 2-phased recovery. In particular, we show that if Q is required to be closed in the synthesized program, then the problem of strict and relaxed 2-phase recovery can be solved in polynomial-time. Towards this end, we present an algorithm Add_StrictPhasedRecovery in Subsection 6.1 and Add_RelaxedPhasedRecovery in Subsection 6.2. For simplicity of presentation, we assume that $S \subseteq Q$ while describing these algorithms. Subsequently, in Subsection 6.3, we show that these results remain valid even if S is not a subset of Q. Finally, in Subsection 6.4, we provide an interpretation for the closure of Q.

6.1 Synthesizing Strict 2-Phase Recovery with $S \subseteq Q$ and Closure of Q

In this subsection, we propose the algorithm Add_StrictPhasedRecovery to validate the following claim:

Claim Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ be a program with invariant S and recovery specification $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \ \wedge \ (Q \mapsto_{\leq \delta} S)$. There exists a polynomial-time sound and complete solution to Problem Statement 1 in the size of the region graph of $\mathcal{P}$, if $(S \subseteq Q) \wedge (Q$ is closed in $\psi_{\mathcal{P}'})$. ∎

Algorithm sketch. Intuitively, our algorithm works as follows. In Step 1, we transform the input program into a *region graph* [2] (described below). In Step 2, we isolate the set of states from where $SPEC_{\overline{bt}}$ may be violated. In Steps 3 and 4, we ensure that any computation of $\mathcal{P}'$ that starts from a state in $\neg S' - Q$ (respectively, $Q - S'$) reaches a state in Q (respectively, S') within θ (respectively, δ) time units. In Step 5, we ensure the closure of fault-span and deadlock freedom of invariant. We repeat Steps 3-4 until a fixpoint is reached. Finally, in Step 6, we transform the resultant region graph back into a real-time program.

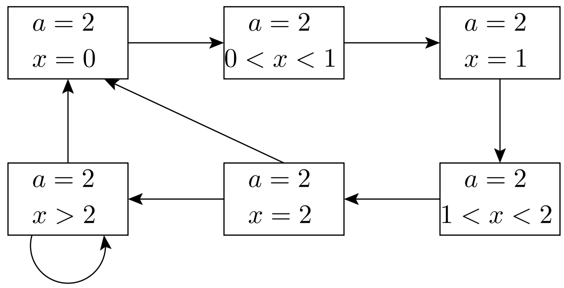

Fig. 2 An example of a region graph.

Assumption 2 Let $\overline{\alpha} = (\sigma_0, \tau_0) \to (\sigma_1, \tau_1) \to \cdots (\sigma_n, \tau_n)$ be a computation prefix where $\sigma_0, \sigma_n \in S$ and $\sigma_i \notin S$ for all $i \in \{1..n-1\}$. Only for simplicity of presentation, we assume that the number of occurrence of faults in $\overline{\alpha}$ is one. Precisely, we assume that in $\overline{\alpha}$, if (σ_0, σ_1) is a fault transition, then no faults occur outside the program invariant. In our previous work [11], we have shown how to deal with cases where multiple faults occur in a computation when adding bounded response properties. The same technique can be applied while preserving soundness and completeness of the algorithm Add_StrictPhasedRecovery in this paper. We discuss this extension in Remark 4 after the Add_StrictPhasedRecovery algorithm. Furthermore, notice that the proofs of Theorems demonstrating NP-hardness of ordered-strict, strict and relaxed 2-phased recovery depend only on the occurrence of one fault. ∎

Region Graph. Real-time programs can be analyzed with the help of an equivalence relation of finite index on the set of states [2]. 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 $\mathcal{P}$ that involve x, where $c_x = 0$ if x does not occur in any clock constraints of $\mathcal{P}$. 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 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 $\tau' < \tau$. Figure 2 shows the region graph of the following guarded command for clock variable x and discrete variable a:

$$(a = 2) \ \wedge \ (x \geq 2) \qquad \xrightarrow{x:=0} \qquad \textbf{skip};$$

Using the region equivalence relation, we construct the *region graph* of $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ (denoted $R(\mathcal{P}) = \langle S^r_\mathcal{P}, \psi^r_\mathcal{P} \rangle$) as follows. 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 $\psi_\mathcal{P}$.

We now describe the algorithm Add_StrictPhasedRecovery in detail:

– (*Step 1*) First, we use the above technique to transform the input program $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ into a region graph $R(\mathcal{P}) = \langle S^r_\mathcal{P}, \psi^r_\mathcal{P} \rangle$. To this end, we invoke the procedure ConstructRegionGraph as a black box (Line 1). We let this procedure convert state predicates and sets of transitions in $\mathcal{P}$ (e.g., S and $\psi_{\mathcal{P}}$) to their corresponding region predicates and sets of edges in $R(\mathcal{P})$ (e.g., S^r and $\psi^r_{\mathcal{P}}$). Precisely, a *region predicate* U^r with respect to a state predicate U is the set $U^r = \{(s,\rho) \mid \exists(s,\nu) : ((s,\nu) \in U \ \wedge \ \nu \in \rho)\}$.

– (*Step 2*) In order to ensure that the synthesized program does not violate $SPEC_{\overline{bt}}$, we identify the set ms of regions from where a computation may reach a transition in $SPEC_{bt}$ by taking fault transitions alone (Line 2). Next (Line 3), we compute the set mt of edges, which contains:

  1. edges that directly violate safety (i.e., $SPEC^r_{bt}$), and
  2. edges whose target region is in ms (i.e., edges that lead a computation to a state from where safety may be violated by faults alone).

  Since the program does not have control over occurrence of faults, we remove the set ms from the region predicate T^r_1, which is our initial estimate of the fault-span (Line 4). Likewise, in Step 3, we will remove mt from the set of program edges $\psi^r_{\mathcal{P}}$ when recomputing program transitions.

– (*Step 3*) In this step, we add recovery paths to $R(\mathcal{P})$ so that $R(\mathcal{P})$ satisfies $\neg S' \mapsto_{\leq\theta} Q$ and $Q \mapsto_{\leq\delta} S'$. To this end, we first recompute the set $\psi_{\mathcal{P}_1}$ of program edges (Line 7) by including:

  1. existing edges that start and end in S^r_1, and
  2. new *recovery edges* that originate from regions in $T^r_1 - Q^r$ (respectively, $Q^r - S^r_1$) and terminate at regions in T^r_1 (respectively, Q) such that the time-monotonicity condition is met.

  We exclude the set mt from $\psi^r_{\mathcal{P}_1}$ to ensure that these recovery edges do not violate $SPEC_{\overline{bt}}$. Notice that the algorithm allows arbitrary clock resets during recovery. If such clock resets are not desirable, one can rule them out by including them as bad transitions in $SPEC_{bt}$.

  After adding recovery edges, we invoke the procedure Add_BoundedResponse (Line 8) with parameters $T^r_1 - Q^r$, Q^r, and θ to ensure that $R(\mathcal{P})$ indeed satisfies the bounded response property $\neg S \ \mapsto_{\leq\theta} \ Q$. The properties of the procedure Add_BoundedResponse (first proposed in [10]) are the following:

  – By Assumption 1, there is a variable that is reset when the program reaches a state in $\neg S \wedge \neg Q$. Let t_1 denote that variable. Clearly, when the program moves from a state in Q to a state in $T_1 - Q$, t_1 is reset to 0. Add_BoundedResponse utilizes this clock variable to compute delay required to reach a state in Q.
  – for each state σ in $T_1 - Q$, it includes the set of transitions that participate in forming the computation that starts from σ and reaches a state in Q with smallest possible time delay, if the delay is less than θ, and
  – the regions made unreachable by this procedure (returned as the set ns) cannot be present in any solution that satisfies $\neg S_1 \mapsto_{\leq\theta} Q$.

  The procedure may optionally include additional computations, provided they preserve the corresponding bounded response property. Thus, since 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 4*) Similar to Step 3, in Line 10, the algorithm utilizes clock variable, say t_2, which gets reset when $Q - S_1$ becomes true and ensures that $R(\mathcal{P})$ satisfies $Q \mapsto_{\leq\delta} S_1$.

Algorithm 1 Add_StrictPhasedRecovery

Input: A real-time program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ with invariant S, fault transitions f, bad transitions $SPEC_{bt}$, intermediate recovery predicate Q s.t. $S \subseteq Q$, recovery time δ, and intermediate recovery time θ.

Output: If successful, a fault-tolerant real-time program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$.

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: $ms \; := \; \{r_0 \mid \exists r_1, r_2 \cdots r_n \; : \; (\forall j \mid 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \quad \wedge \quad (r_{n-1}, r_n) \in SPEC^r_{bt}\}$;
3: $mt \; := \; \{(r_0, r_1) \mid (r_1 \in ms) \; \vee \; ((r_0, r_1) \in SPEC^r_{bt})\}$;
4: $T^r_1 \; := \; S^r_{\mathcal{P}} - ms$;
5: **repeat**
6: $\quad T^r_2, \; S^r_2 \; := \; T^r_1, \; S^r_1$;
7: $\quad \psi^r_{\mathcal{P}_1} \; := \; \psi^r_{\mathcal{P}} | S^r_1 \; \cup \; \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (T^r_1 - Q^r) \; \wedge \; (s_1, \rho_1) \in T^r_1 \; \wedge$
$\qquad \exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \; : \; (\exists \lambda \subseteq X \; : \; \rho_1 = \rho_2[\lambda := 0])\} \; \cup$
$\qquad \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (Q^r - S^r_1) \; \wedge \; (s_1, \rho_1) \in Q^r \; \wedge$
$\qquad \exists \rho_2 \mid \rho_2$ is a time-successor of $\rho_0 \; : \; (\exists \lambda \subseteq X \; : \; \rho_1 = \rho_2[\lambda := 0])\} \; - \; mt$;
8: $\quad \psi^r_{\mathcal{P}_1}, \; ns$:= Add_BoundedResponse($\langle S^r_{\mathcal{P}}, \psi^r_{\mathcal{P}_1} \rangle$, $T^r_1 - Q^r$, Q^r, θ);
9: $\quad T^r_1 \; := \; T^r_1 \; - \; ns$;
10: $\quad \psi^r_{\mathcal{P}_1}, \; ns$:= Add_BoundedResponse($\langle S^r_{\mathcal{P}}, \psi^r_{\mathcal{P}_1} \rangle$, $Q^r - S^r_1$, S^r_1, δ);
11: $\quad T^r_1, \; Q^r \; := \; T^r_1 \; - \; ns, \; Q^r \; - \; ns$;
12: $\quad$ **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: $\qquad T^r_1 := T^r_1 - \{r_0\}$;
14: $\quad$ **end while**
15: $\quad$ **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: $\qquad S^r_1 := S^r_1 - \{r_0\}$;
17: $\quad$ **end while**
18: $\quad$ **if** $(S^r_1 = \{\} \; \vee \; T^r_1 = \{\})$ **then**
19: $\qquad$ **print** `‘‘no fault-tolerant program exists’’`; **exit;**
20: $\quad$ **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';

– (*Step 5*) Since we remove the set ns of regions from T^r_1, we need to ensure that T_1 is closed in f. Thus, we remove regions from where a sequence of fault edges can reach a region in ns (Lines 12-14). Next, due to the possibility of removal of some regions and edges in the previous steps, the algorithm ensures that the region graph $\langle S^r_{\mathcal{P}}, \psi^r_{\mathcal{P}_1} \rangle$ does not have deadlock regions in the region invariant S^r_1 (Lines 15-17). Precisely, we say that a region (s_0, ρ_0) of region graph $R(\mathcal{P}) = \langle S^r_{\mathcal{P}}, \psi^r_{\mathcal{P}} \rangle$ is a *deadlock region* in region predicate U^r iff for all regions $(s_1, \rho_1) \in U^r$, there does not exist an edge of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1) \in \psi^r_{\mathcal{P}}$. Deadlock freedom in the region graph is necessary, as the constraint $C4$ in the Problem Statement 1 does not allow the algorithm to introduce new finite or time-divergent computations to the input program. If the removal of deadlock regions and regions from where the closure of fault-span is violated results in empty invariant or fault-span, the algorithm declares failure (Lines 18-20).

– (*Step 6*) Finally, upon reaching a fixpoint, we transform the resulting region graph $\langle S^r_{\mathcal{P}}, \psi^r_{\mathcal{P}_1} \rangle$ back into a real-time program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ by invoking the procedure ConstructRealTimeProgram. In fact, the program $\mathcal{P}'$ is returned as the final synthesized fault-tolerant program. Note that since a region graph is a time-abstract *bisimulation* [2], we will not lose any behaviors in the reverse transformation.

Remark 4 The algorithm Add_StrictPhasedRecovery deals with the case where only a single fault occurrence is considered. For the case where multiple faults (with a known bound on the maximum number of faults) occur, we need to make changes to Add_BoundedResponse to take multiple faults into account. Specifically, in Step 3 of the above algorithm, we noted that Add_BoundedResponse computes the shortest possible delay from every region to reach the destination predicate (i.e., predicate B in $(A \mapsto_{\leq\theta} B)$). This delay computation needs to be changed if multiple faults can occur. To achieve this, intuitively, we consider 'layers' of region graph. The top layer corresponds to the case where one fault has occurred. The next lower layer corresponds to the case where two faults have occurred and so on. The bottom layer corresponds to the case where N faults have occurred where N is the maximum number of permitted faults. For simplicity of presentation, in the subsequent discussion let $N = 2$ thereby limiting the number of layers to 2. Computing delay in the bottom layer is straightforward since no new faults can occur. Now, for the top layer, if a fault can take the program from region R_1 to region R_2, we include an edge from 'R_1 from top layer' to 'R_2 from bottom layer'. Now, computing delays in top layer needs to incorporate such fault transitions into account. Specifically, delay associated with 'R_1 from top layer' must be at least equal to the delay associated with 'R_2 from bottom layer' + 'delay associated with the fault transition (if any)'. With these changes to Add_BoundedResponse, it is possible to permit multiple faults in Add_StrictPhasedRecovery. The details of the changes required to modify Add_BoundedResponse to deal with multiple faults can be found in [11]. Also, [11] also shows that in most cases, if the bound N is not known then computing adding bounded response property is not possible.

We now show that the algorithm Add_StrictPhasedRecovery is *sound* in the sense that any program that it synthesizes is correct-by-construction. We also show that the algorithm is *complete* in the sense that if it fails to synthesize a solution then no other correct solution exists.

Theorem 7 *The Algorithm* Add_StrictPhasedRecovery *is sound.*

Proof We show that the algorithm satisfies the constraints of Problem Statement 1. Let t_1 and t_2 denote the variables used by Add_BoundedResponse by Assumption 1 (cf. Lines 8 and 10). We proceed as follows:

1. (*Constraints* $C1$ *and* $C2$) By construction, correctness of these constraints trivially follows.
2. (*Constraint* $C3$) 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 of no concern 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, say r_0, in S'^r, 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 an integer. 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, we need to show that every computation of $\psi_{\mathcal{P}'}[]f$ that starts from a state in T' reaches a state in Q and subsequently a state in S' within θ and δ time units, respectively. 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'. If $\sigma_0 \in S'$ a single fault transition may take $\overline{\sigma}$ to $T' - S'$ and by Assumption 2, none of the subsequent transitions in $\overline{\sigma}$ are in f. Thus, $\overline{\sigma}_1 = (\sigma_1, \tau_1) \rightarrow (\sigma_2, \tau_2) \rightarrow \cdots$ is a computation of $\psi_{\mathcal{P}'}$ where $\sigma_1 \in T' - S'$. In the algorithm, when the repeat-until loop terminates, by construction, (1) $\overline{\sigma}_1$ reaches a state, say σ_k, in Q where $\tau_k - \tau_1 \leq \theta$ (cf. Lines 7-8), and (2) from σ_k, $\overline{\sigma}$ reaches a state in S' within δ (cf. Line 10). This also implies that in the presence of faults as well, our algorithm does not introduce time-convergent computations to $\mathcal{P}'$, as $\overline{\sigma}$ eventually reaches a state in S'. ∎

Theorem 8 *The Algorithm* Add_StrictPhasedRecovery *is complete.*

Proof The proof of completeness is based on the observation that if any state is removed, then it must be removed, i.e., there is no fault-tolerant program that meets the constraints of Problem Statement 1 and includes this state. For example, in the computation of ms, if (σ_0, σ_1) is a fault transition and violates safety then state σ_0 must be removed (i.e., is made unreachable). Likewise, ms includes states from where execution of faults alone violates safety. Hence, they must be removed. In Line 7, we compute the input program that includes all possible transitions that may be used in the final program. Due to constraint $C2$ of the Problem Statement 1, any transition that begins in the invariant must be a transition of the fault-intolerant program. Due to closure of Q in the sufficient condition, any transition from Q (precisely $Q - S$ since states in S are already handled) must end in Q. And, due to closure of fault-span, any transition that begins in T (precisely $T - Q$) 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 response 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 fault-intolerant program are illegitimate with respect to Problem Statement 1. Therefore, the algorithm Add_StrictPhasedRecovery is complete. ∎

Theorem 9 *The complexity of Algorithm* Add_StrictPhasedRecovery *is in polynomial-time in the size of the input program's region graph*[3]*.*

[3] We note that the algorithm for constructing a region graph is in PSPACE [2]. In fact, the reachability problem in real-time programs is PSPACE-complete [2]. Thus, the complexity of our algorithm in the size of the given fault-intolerant program is in PSPACE. This analysis holds for other algorithms presented in latter sections if the paper as well. We also note that a region graph is not the most efficient data structure to analyze real-time programs. We use region graphs in this paper, as our goal is complexity analysis and not efficiency in implementation. For more efficient implementation, one can use symbolic data structures such as *zone graphs*.

Proof This follows from the fact that the complexity of each line is polynomial in the size of the region graph and that the number of iterations for any loop is polynomial in the size of the region graph. ∎

Remark 5 The algorithm Add_StrictPhasedRecovery can be easily revised for the case where n-phase recovery is desired. Specifically, if Q_1 and Q_2 are intermediate predicates and they are required to be closed in the synthesized program then Line 7 of code needs to be revised so that we do not include transitions that violate Q_1 and transitions that do not violate Q_2. Also, Add_RelaxedPhasedRecovery (in Section 6.2) can be extended in a similar fashion.

6.1.1 Example (cont'd)

We now demonstrate how the algorithm Add_StrictPhasedRecovery synthesizes a fault-tolerant version of $\mathcal{TC}$. In the recovery specification of $\mathcal{TC}$ (cf. in Subsection 3.2), the invariant predicate $S_{\mathcal{TC}}$ and intermediate recovery predicate $Q_{\mathcal{TC}}$ were disjoint. Now, let the intermediate recovery predicate be:

$$Q_{new} = S_{\mathcal{TC}} \ \cup \ Q_{\mathcal{TC}}.$$

In other words, after the occurrence of faults, the recovery specification requires that either both signals turn red within 3 time units and then return to the normal behavior within 7 time units, or, the system reaches a state in $S_{\mathcal{TC}}$ within 3 time units. Since, $S_{\mathcal{TC}} \subseteq Q_{new}$, we can apply the Algorithm Add_BoundedPhasedRecovery to transform $\mathcal{TC}$ into a fault-tolerant program $\mathcal{TC}'$. We note that due to many symmetries in $\mathcal{TC}$ and the complex structure of the algorithm, we only present a highlight of the process of synthesizing $\mathcal{TC}'$.
First, observe that in Step 2 of the algorithm, $ms = \{\}$ and $mt = SPEC_{bt_{\mathcal{TC}}}$. In Step 3, consider a subset of $T_1 - Q_{new}$ where $(sig_0 = sig_1 = R) \wedge (z_0, z_1 \leq 1)$. This predicate is reachable by a single occurrence of (for instance) F_0 from an invariant state where $(sig_0 = sig_1 = R) \wedge (z_0 > 1) \wedge (z_1 \leq 1)$. After adding legitimate recovery transitions (Line 7), the invocation of Add_BoundedResponse (Line 8) results in addition of the following recovery action:

$$\mathcal{TC}5_i \ :: \quad (sig_0 = sig_1 = R) \ \wedge \ (z_0, z_1 \leq 2) \ \wedge \ (t_1 \leq 2) \quad \longrightarrow \quad \textbf{wait};$$

for all $i \in \{0, 1\}$. This action enforces the program to take delay transitions so that the program reaches a state in Q where $(sig_0 = sig_1 = R) \wedge (z_0, z_1 > 1)$. Recall that the clock variable t_1 is used byAdd_BoundedResponse to keep track of the time elapsed since $\neg S$ holds (see Assumption 1).

One may notice that although it is perfectly legitimate to wait up to 3 time units inside $T_1 - Q_{new}$, as $\theta = 3$, the action $\mathcal{TC}5$ lets the program wait only for 2 time units. This is because Add_BoundedResponse first includes computations with the smallest possible time delay and then *optionally* includes additional computations to increase the level of non-determinism. In this context, additional computations may be constructed by the following actions for sig_0:

$$\begin{array}{lll}
\mathcal{TC}6_0 :: & (sig_0 = sig_1 = R) \ \wedge \ (z_0, z_1 \leq 1) \ \wedge \ (t_1 \leq 1) \quad \xrightarrow{\{x_0\}} & (sig_0 \ := \ G); \\
\mathcal{TC}7_0 :: & (sig_0 = G) \wedge (sig_1 = R) \wedge & \\
 & (x_0 = 1) \ \wedge \ (z_0, z_1 \leq 2) \ \wedge \ (t_1 \leq 2) \quad \xrightarrow{\{y_0\}} & (sig_0 \ := \ Y); \\
\mathcal{TC}8_0 :: & (sig_0 = G) \wedge (sig_1 = R) \wedge & \\
 & (x_0 \leq 1) \ \wedge \ (z_0, z_1 \leq 2) \ \wedge \ (t_1 \leq 2) \quad \longrightarrow & \textbf{wait};
\end{array}$$

Notice that while executing recovery action $\mathcal{TC}6_0$ results in reaching another state in $T_1 - Q_{new}$, execution of actions $\mathcal{TC}7_0$ and $\mathcal{TC}8_0$ result in reaching a state in invariant $S_{\mathcal{TC}}$, which is clearly in Q_{new} as well.

Now, consider the case where $\mathcal{TC}$ is in a state where $(sig_0 = G) \wedge (sig_1 = R) \wedge (x_0 = 1) \wedge (z_0, z_1 \leq 2)$. In this case, one may argue that $\mathcal{TC}$ has the option of executing action $\mathcal{TC}3_1$ and reaching a state where $sig_0 = sig_1 = G$, which is clearly a violation of safety specification $SPEC_{\overline{bt}_{\mathcal{TC}}}$. However, since we remove the set mt from $\psi_{\mathcal{P}_1}$ (Line 7), action $\mathcal{TC}3_i$ would be revised as follows:

$$\mathcal{TC}3_i{::} \quad (sig_i = R) \wedge (z_j \leq 1) \wedge (sig_j = R) \quad \xrightarrow{\{x_i\}} \quad (sig_i := G);$$

for all $i \in \{0, 1\}$ where $j = (i + 1) \mod 2$. In other words, the algorithm strengthens the guard of $\mathcal{TC}1_i$, such that in the presence of faults, a signal turns green only when the other one is red.

In Step 4, consider the state predicate $Q_{new} - S_{1_{\mathcal{TC}}} = (sig_0 = sig_1 = R) \wedge (z_0, z_1 > 1)$. Similar to Step 3, the algorithm adds recovery paths with the smallest possible time delay, which is the following action for either $i \in \{0, 1\}$:

$$\mathcal{TC}9_i{::} \quad (sig_i = sig_j = R) \wedge (z_i, z_j > 1) \quad \xrightarrow{\{z_i\}} \quad \textbf{skip};$$

It is straightforward to verify that by execution of $\mathcal{TC}9_i$, the program reaches the invariant $S_{\mathcal{TC}}$ from where the program behaves normally. Similar to Step 3, the procedure Add_BoundedResponse may include the following additional actions:

$$\begin{array}{llll}
\mathcal{TC}10_i{::} & (sig_i = sig_j = R) \wedge (z_i, z_j > 1) \wedge (t_2 \leq 7) & \xrightarrow{\{x_i\}} & (sig_i := G); \\
\mathcal{TC}11_i{::} & (sig_i = sig_j = R) \wedge (z_i, z_j > 1) \wedge (t_2 \leq 7) & \xrightarrow{\{y_i\}} & (sig_i := Y); \\
\mathcal{TC}12_i{::} & (sig_i = sig_j = R) \wedge (z_i, z_j > 1) \wedge (t_2 \leq 7) & \longrightarrow & \textbf{wait};
\end{array}$$

One may notice that action $\mathcal{TC}11_i$ adds a strange behavior to $\mathcal{TC}$ by allowing a signal to change phase from red to yellow. Our algorithm allows addition of such recovery action, since it does not violate the safety specification $SPEC_{\overline{bt}_{\mathcal{TC}}}$. One may enforce the algorithm not to add such actions by simply adding the transitions in the set $\{(\sigma_0, \sigma_1) \mid \exists i \in \{0, 1\} : (sig_i(\sigma_0) = R) \wedge (sig_i(\sigma_1) = Y)\}$ to $SPEC_{bt_{\mathcal{TC}}}$. In fact, we expect that our synthesis techniques have the potential to identify missing properties in cases where the specification is incomplete.

In the context of $\mathcal{TC}$, in Step 5, the algorithm removes states from neither the fault-span nor the invariant, as $ns = \{\}$, and, hence, the algorithm finds the final solution in one iteration of the repeat-until loop.

6.2 Synthesizing Relaxed 2-Phase Recovery with $S \subseteq Q$ and Closure of Q

In this section, we propose the algorithm Add_RelaxedPhasedRecovery to validate the following claim:

Claim Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ be a program with invariant S and recovery specification $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \wedge (\neg S \mapsto_{\leq \delta} S)$. There exists a polynomial-time sound and complete solution to Problem Statement 1 in the size of the region graph of $\mathcal{P}$, if $(S \subseteq Q) \wedge (Q \text{ is closed in } \psi_{\mathcal{P}'})$. ∎

Algorithm 2 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 $\psi_{\mathcal{P}'}$.

// *This algorithm is obtained by changing the following lines from Algorithm 1*

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, δ);
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, θ));

The algorithm Add_RelaxedPhasedRecovery (cf. Algorithm 2) is also based on Assumption 2. Since this algorithm reuses most of Add_StrictPhasedRecovery, 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*) This step is also identical to that in Algorithm 1 and it constructs ms and mt.
- (*Step 3: Adding* $(Q \mapsto_{\leq \delta} S)$) In this step, we first recompute the set $\psi^r_{\mathcal{P}_1}$ of program edges (Line 7) that is identical to Algorithm 1. Then, we use $\psi^r_{\mathcal{P}_1}$ on Line 7 to invoke the procedure Add_BoundedResponse (from [10]) to add $(Q \mapsto_{\leq \delta} S)$. As mentioned in Subsection 6.1, 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 4: 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 $T^r_1 - Q^r$ to r now reach r_1. All other edges (edges reaching r from another region 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 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 5 and 6: Repeat if needed or construct synthesized program*) These steps are identical to that in Algorithm 1.

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 10 *The Algorithm* Add_RelaxedPhasedRecovery *is sound.*

Proof To prove this theorem, we show that the algorithm Add_RelaxedPhasedRecovery satisfies the constraints of Problem Statement 1 when instantiated with relaxed 2-phase recovery. Let t_1 and t_2 denote the variables used by Add_BoundedResponse by Assumption 1 (cf. Lines 8 and 10). We proceed as follows:

1. (*Constraints* $C1$ *and* $C2$) By construction, correctness of these constraints trivially follows.
2. (*Constraint* $C3$) 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, $\mathcal{P}' \models_{S'} SPEC$. The proof of this subgoal is identical to that of Theorem 7.
   - 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 in 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.

Theorem 11 *The Algorithm* Add_RelaxedPhasedRecovery *is complete.*

Proof The proof of completeness is identical to that of Theorem 8.

Theorem 12 *The complexity of Algorithm* Add_RelaxedPhasedRecovery *is in polynomial-time in the size of the input program's region graph.* ∎

Proof The proof of completeness is similar to that of Theorem 9.

We note that applying Algorithm Add_RelaxedPhasedRecovery on $\mathcal{TC}$ results in deriving $\mathcal{TC}'$ identical to the one synthesized in Subsection 6.1.1. This is due to the fact that in both cases, we assumed $S \subseteq Q$. If this requirement is eliminated, one can derive different fault-tolerant programs.

6.3 Synthesizing Strict and Relaxed 2-Phase Recovery with $S \not\subseteq Q$ and Closure of Q

In this section, we show that the problem of synthesizing strict and relaxed 2-phase recovery can be solved in polynomial-time, if Q is required to be closed in the synthesized program even if $S \subseteq Q$ is not satisfied. In particular, we show that the algorithms Add_StrictPhasedRecovery and Add_RelaxedPhasedRecovery can be applied in this context by changing their parameters appropriately. To show this for strict 2-phase recovery, we make the following observation.

Observation 3 *Let state predicates S and Q be closed in program $\mathcal{P}$. We have:*
$\mathcal{P}$ *satisfies* $(\neg S \mapsto_{\leq\theta} Q) \wedge (Q \mapsto_{\leq\delta} S)$
iff
$Q \cap S$ *is closed in* $\mathcal{P}$ *and* $\mathcal{P}$ *satisfies* $(\neg S \mapsto_{\leq\theta} Q) \wedge (Q \mapsto_{\leq\delta} Q \cap S)$.

From the above observation, it follows that even if $S \subseteq Q$ is not satisfied, we can utilize algorithm Add_StrictPhasedRecovery, where S is instantiated with $Q \cap S$. One application of Observation 3 is in providing *graceful degradation*. To illustrate this, consider the case where the invariant S is of the form $S_1 \cup S_2$, where $S_1 \cap S_2 = \phi$ and the program provides ideal behavior in S_1 and behavior with reduced functionality in S_2. And, let Q be a superset of $S2$. Thus, $S \subseteq Q$ does not hold. When Add_StrictPhasedRecovery is applied in this context, it would mean that after the occurrence of a fault, the program must provide an acceptable behavior within time θ. And, subsequently, it must provide reduced functionality within time δ.

Observation 4 *Let state predicates S and Q be closed in program $\mathcal{P}$. We have:*
$\mathcal{P}$ *satisfies* $(\neg S \mapsto_{\leq\theta} Q) \wedge (\neg S \mapsto_{\leq\delta} S)$*, where* $\delta \geq \theta$
iff
$Q \cap S$ *is closed in* $\mathcal{P}$ *and* $\mathcal{P}$ *satisfies* $(\neg S \mapsto_{\leq\theta} Q) \wedge (\neg S \mapsto_{\leq\delta} Q \cap S)$.

From the above observation, if $\delta \geq \theta$, then we can utilize Add_RelaxedPhasedRecovery, where S is instantiated with $Q \cap S$. Due to symmetry of the above observation in terms of Q and S, if $\delta \leq \theta$, then we can utilize Add_RelaxedPhasedRecovery, where Q is instantiated with $Q \cap S$.

Based on Observations 3 and 4, it follows that if Q is required to be closed in the synthesized program then the problems of synthesizing strict or relaxed 2-phase recovery can be solved in polynomial-time even if $S \subseteq Q$ is not satisfied.

6.4 Interpretation of Closure of Q

One main observation from the results in Section 5 and Subsections 6.1 and 6.2 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 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. One instance of latter corresponds to the case where

Algorithm 3 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 2

$7 : \psi^r_{\mathcal{P}_1} \ := \ \psi^r_{\mathcal{P}} | S^r_1 \ \cup \ \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \in (T^r_1 - S^r) \ \wedge \ (s_1, \rho_1) \in T^r_1 \ \wedge$
$\quad \exists \rho_2 \mid \rho_2 \text{ is a time-successor of } \rho_0 \ : \ (\exists \lambda \subseteq X \ : \ \rho_1 = \rho_2[\lambda := 0])\} \ \ - mt;$

$10 : \psi^r_{\mathcal{P}_1}, \ ns \ := \text{Add\_BoundedResponse}((\langle S^r_{\mathcal{P}}, \psi^r_{\mathcal{P}_1} \rangle), T^r - S^r_1, S^r_1, \theta);$

upon occurrence of fault, the program must take an action that *records* the fault before ideal behavior is resumed. Specifically, in such an instance, the predicate Q would correspond to the case where the program is recording the occurrence of fault. Clearly, in this case, Q is not closed since the program is not expected to record faults during fault-free behavior. The results in this paper show that complexity for such a problem is high. Thus, one interpretation of the result about closure of Q is as follows: If Q is used to characterize *acceptable* behavior that is reached quickly then the complexity of adding 2-phase recovery will be low. However, if Q is used to characterize *recording of fault* (or other such behavior that does not occur during fault-free behavior) then the complexity increases substantially.

7 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 2 from Subsection 6.2. 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 and 2*) These steps are identical to that in Algorithm 2 and they construct the region graph $R(\mathcal{P})$, ms and mt.
– (*Step 3: 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_1^r.

- (*Step 4: Adding* $(\neg S \mapsto_{\leq\theta} S)$) This task is achieved by calling Add_BoundedResponse, so that 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_1^r. While the addition of the second bounded response property is possible for graceful 2-phase recovery, for reasons discussed after Theorem 13, it is not possible for relaxed 2-phase recovery.
- (*Step 5 and 6*) These steps are identical to those in Algorithm 2.

Theorem 13 *The Algorithm* Add_GracefulPhasedRecovery *is sound and complete.*

Proof The proof of soundness is similar to that of Theorems 10. In particular, regarding soundness, for constraints $C1$ and $C2$ as well as the correctness of the synthesized program in the absence of faults, the same argument as given in Theorem 10 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 10 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. ∎

Theorem 14 *The complexity of Algorithm* Add_GracefulPhasedRecovery *is in polynomial-time in the size of the input program's region graph.* ∎

Proof The proof of completeness is similar to that of Theorem 9.

Proof of Theorem 4. This theorem states that the problem of transforming a fault-intolerant program to a fault-tolerant program that provides strict 2-phase recovery where $S \subseteq Q$ and $\theta = \infty$ can be solved in polynomial-time. Recall that this problem requires us to add $(\neg S \mapsto_{\leq\infty} Q)$ and $(Q \mapsto_{\leq\delta} S)$. Furthermore, since $S \subseteq Q$, this is equivalent to adding $(\neg S \mapsto_{\leq\infty} S)$ and $(Q \mapsto_{\leq\delta} S)$. Observe that this is an instance of graceful 2-phase recovery. Hence, Theorem 4 follows from Theorem 14. ∎

Next, we discuss the main differences between the two algorithms. We identify the main reason that permits the 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.

7.1 Example (cont'd)

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 $(Q_{\mathcal{TC}} \mapsto_{\leq 3} S_{\mathcal{TC}})$. Recall that variable t_1 is used whenever 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 (see Assumption 1). 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 $Q_{\mathcal{TC}}$ is not affected by the new transitions included on Line 7. Hence, the program adds the following 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. Note that Add_BoundedResponse can add additional transitions that do not violate the required timing constraints. However, this addition must be done after ensuring the second timing constraint (Line 10).

Then, the program adds $(\neg S_{\mathcal{TC}} \quad \mapsto_{\leq 7} \quad S_{\mathcal{TC}})$. 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.

8 Related Work

Discrete controller synthesis (DCS) was first introduced by Ramadge and Wonham [37]. The objective in DCS is to find a language C (the *controller*) for a *plant* represented by language P and a specification language S, such that $P \cap C \subseteq S$. Intuitively, in DCS, the goal is to constrain the actions of the plant by a controller, such that the behavior of the controlled plant always meets safety and/or reachability conditions required in the specification. Controller synthesis has extensively been studied from different perspectives. Examples include, on-the-fly controller synthesis [41], controller synthesis with partial observability [32], distributed controller synthesis [38], and optimal supervisory control [31]. Timed controller synthesis was first introduced by Maler, Pnueli, and Sifakis [34]. This work was later extended in [18, 22, 6, 7]. The problem of synthesizing controllers for bounded response properties is considered in [33]. Our work in this paper is different from [33] in that we revise a given program in the presence of faults with respect to bounded response properties, but the authors in [33] synthesize an automaton that satisfies a set of such properties. Also, our focus is on the impact of the relation between predicates involved in the bounded response properties on the complexity of synthesis.

The idea of transforming a fault-intolerant system into a fault-tolerant system using controller synthesis was first developed by Chao and Lim [19]. Similar to the model in this paper, Chao and Lim consider faults as a system malfunction and failures as something that should not occur in any execution. Their control objective is a set of states that should be reachable by controllable actions or what they define as *recurrent events*. Also, Girault and Rutten [26], demonstrate the application of discrete controller synthesis in automated addition of fault-tolerance in the context of untimed systems. They model different types of faults (e.g., processor crash, Byzantine faults, value corruption) by uncontrollable actions in a labeled transition system (LTS). They show that given a fault-intolerant program as a plant, discrete controller synthesis can automatically add fault-tolerance to the synchronous product of the plant and the fault model LTS with respect to invariance and reachability constraints.

We note that the notion of dependability and in particular fault-tolerance involves features beyond just invariance and reachability. One such feature is *bounded-time recovery*, where a program returns to its normal behavior when its state is perturbed by the occurrence of faults. Recovery is an essential building block in fault-tolerant systems and it is the focus of this paper. In all the aforementioned papers, and in particular in [26], which is in spirit close to our work, the recovery mechanism must be given as input to the DCS algorithm. Thus, the key difference between our work in this paper and the methods in [26, 18, 22, 6, 7, 19, 34, 37, 41, 32, 38, 31] is the fact that we automatically synthesize recovery paths based on the the given 2-phase recovery specification.

Automated addition of single-phase recovery has been studied from different perspectives. In [11], we presented an algorithm for addition of fault-tolerance and single-phase recovery to real-time programs, where multiple faults can occur outside the program's normal behavior. One can extend the polynomial-time algorithms presented in this paper using the technique described in [11] to handle multiple occurrences of faults. A distributed algorithm for adding single-phase recovery to centralized untimed programs was introduced in [17], where the state space of the given fault-intolerant program is distributed over a network or cluster of workstations. In [14], we showed that algorithmic addition of single-phase recovery to distributed programs is NP-complete even in the absence of faults. We present an efficient and effective BDD-based heuristic in [12] for adding single-phase recovery to distributed programs in the presence of faults. This technique is implemented in the tool Sycraft [15]. Finally, addition of single-phase recovery with respect to different classes of faults (known as *multi-tolerance*) was studied in [30].

Automata-theoretic approaches for synthesizing controllers and reactive programs [35] are generally based on the model of two-player games [40]. In such games a program makes moves in response to the moves of its environment. The program and its environment interact through a set of interface variables and, hence, the environment can only update the interface variables. In our model, however, faults can perturb all program variables. Moreover, in a two-player game model, players take turns and the set of states from where the first player can make a move is disjoint from the set of states from

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

Table 2 Other types of 2-phase recovery.

where the second player can move [42]. To the contrary, in our work, fault-tolerance should be provided against faults that can execute from any state. Game-theoretic methods are based on the theory of tree automata [39]. Such an automaton represents the specification of a system. A synthesis algorithm checks the non-emptiness of the automaton, i.e., whether there exists a tree acceptable by the tree automaton. If the tree automaton is indeed nonempty, then the specification is called *realizable* and there exists a model of the synthesized program. Pnueli and Rosner address the problem of synthesizing synchronous open reactive modules in [35]. They generalize their method in [36], by proposing a technique for synthesizing asynchronous reactive modules. In particular, they investigate the problem of synthesizing an asynchronous reactive module that includes only one process and interacts with a non-deterministic environment through Boolean variables. Instances of solving timed games appear in [20, 23].

9 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-phase recovery and characterized their applications in real-world systems (see Table 2). We showed that, in general, the problems of synthesizing ordered-strict, strict, and relaxed 2-phase recovery are NP-complete in the size of the region graph of the given intolerant program. However, the problem for strict and relaxed 2-phase recovery can be solved in polynomial-time (in the region graph), if $S \subseteq Q$ and $\theta = \infty$, or, Q is required to be 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 (in the region graph) even though all other variations are NP-complete. We also identified other subproblems where the problem remains NP-complete or it can be solved in polynomial-time. Appendix A presents a summary of results in a graphical fashion.

Other types of 2-phase recovery are also possible (see Table 2). Other interesting possible values for Q_1 are S and $Q - S$, and, another interesting possible value for Q_2 is $\neg S$. Of these, it is straightforward to observe that the proof of NP-completeness of relaxed 2-phase recovery can be extended to show that synthesizing ordered-relaxed 2-phase recovery is also NP-complete. All the complexity results in this paper are summarized in Figure 3 along with the corresponding theorems that prove those results.

Based on the complexity analysis, we find that the problems of synthesizing strict and relaxed 2-phase recovery are 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. Following our experience with synthesizing *distributed* fault-tolerant programs [12,15], where the problem is NP-complete in the state space, we believe that efficient implementation of such heuristics makes it possible to synthesize real programs in practice. One can also consider the case where a real-time program is subject to different classes of faults and a different type of tolerance and, hence, a different recovery mechanism is required for each fault class. Another interesting problem is to solve the synthesis problem in a compositional fashion. To this end, we are currently extending the BIP component-based framework [9] to add features for reasoning about fault-tolerant components and their interactions.

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. A. Arora. Efficient reconfiguration of trees: A case study in methodical design of non-masking fault-tolerant programs. *Science of Computer Programming*, 1996.
5. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
6. 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.
7. 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.
8. J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2002.
9. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.
10. 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.
11. 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.
12. 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.
13. 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.
14. B. Bonakdarpour and S. S. Kulkarni. Revising distributed UNITY programs is NP-complete. In *Principles of Distributed Systems (OPODIS)*, pages 408–427, 2008.
15. B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*, pages 167–171, 2008.
16. B. Bonakdarpour and S. S. Kulkarni. On the complexity of relaxed and graceful bounded-time 2-phase recovery. In *International Symposium on Formal Methods (FM)*, pages 660–675, 2009.

17. B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Distributed synthesis of fault-tolerant programs in the high atomicity model. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 4838, pages 21–36, 2007.
18. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.
19. 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*, 14(2):348–351, 1998.
20. 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.
21. M. Demirbas, A. Arora, V. Mittal, and V. Kulathumani. A fault-local self-stabilizing clustering service for wireless ad hoc networks. *IEEE Trans. Parallel Distrib. Syst.*, 17(9):912–922, 2006.
22. 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.
23. M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.
24. S. Fortune, J. E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.
25. S. Gilbert, N. A. Lynch, S. Mitra, , and T. Nolte. Self-stabilizing robot formations over unreliable networks. *TAAS*, 4(3), 2009.
26. A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design (FMSD)*, 35(2):190–225, 2009.
27. M. G. Gouda. Multiphase stabilization. *IEEE Trans. Softw. Eng.*, 28(2):200–208, 2002.
28. M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.
29. T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.
30. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.
31. R. Kumar and V. K. Garg. Optimal supervisory control of discrete event dynamicalsystems. *SIAM Journal on Control and Optimization*, 33(2):419–439, 1995.
32. F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions On Automatic Control*, 35(12), December 1990.
33. O. Maler, D. Nickovic, and A. Pnueli. On synthesizing controllers from bounded-response properties. In *Computer Aided Verification (CAV)*, pages 95–107, 2007.
34. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 229–242, 1995.
35. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages (POPL)*, pages 179–190, 1989.
36. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *International Colloqium on Automata, Languages, and Programming (ICALP)*, pages 652–671, 1989.
37. P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
38. K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions On Automatic Control*, 37(11):1692–1708, 1992.
39. W. Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter 4: Automata on Infinite Objects, pages 133–192. Elsevier Science Publishers B. V., Amsterdam, 1990.
40. W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science (STACS)*, pages 1–13, 1995.
41. S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense time systems. In *Formal Methods 1999 (FM)*, pages 233–252, 1999.
42. N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata (CIAA)*, pages 11–22, 2003.

A Graphical Summary of Results for Relaxed and Strict 2-phase Recovery

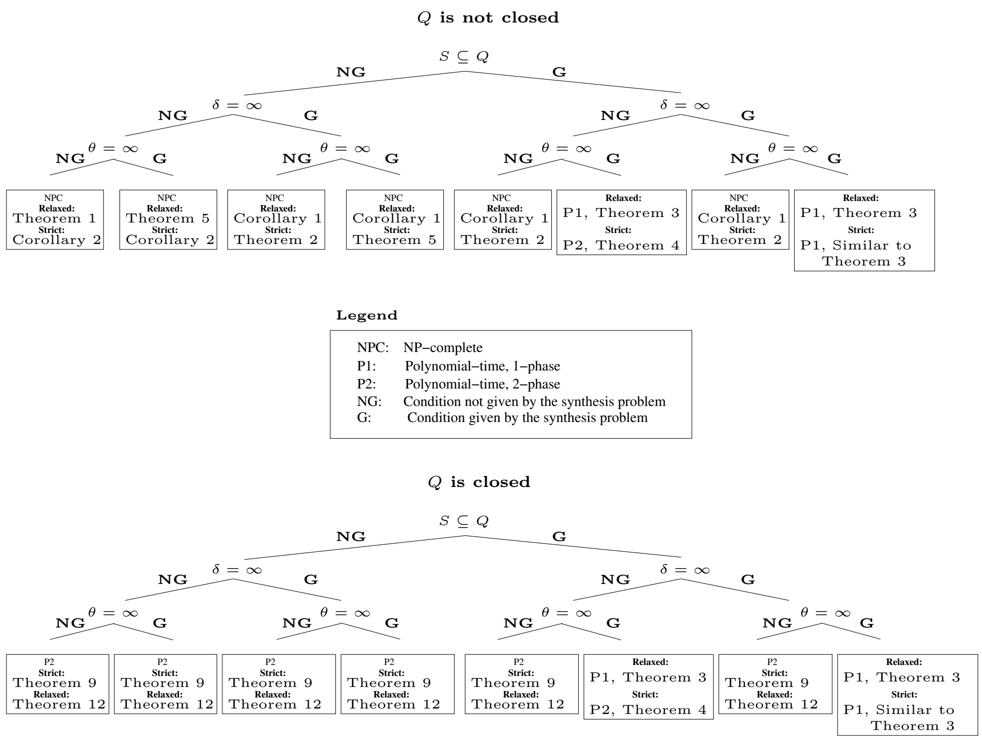

Fig. 3 Summary of Complexity Results