Masking Faults While Providing Bounded-Time Phased Recovery[*]

Borzoo Bonakdarpour and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {borzoo,sandeep}@cse.msu.edu

Abstract. We focus on synthesis techniques for transforming existing fault-intolerant real-time programs to 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 this paper, we show that, in general, synthesizing fault-tolerant real-time programs that provide bounded-time phased recovery is NP-complete. We also characterize a sufficient condition for cases where synthesizing fault-tolerant real-time programs that provide bounded-time phased recovery can be accomplished in polynomial-time in the size of the input program's region graph. Our NP-completeness result also validates our conjecture on complexity of synthesizing hard-masking fault-tolerant programs in our previous work.

Keywords: Fault-tolerance, Real-time, Bounded-time recovery, Phased recovery, Synthesis, Transformation, Formal methods.

1 Motivation

In this paper, we focus on the problem of automated synthesis for real-time systems that provide bounded-time phased recovery in the presence of faults. To illustrate this problem, first, we provide a motivating example to informally describe the idea of bounded-time phased recovery and the concepts of synthesis and fault-tolerance. We also use this example as a running demonstration throughout the paper.

[*] This is an extended version of a paper appeared in the proceedings of FM'08: International Symposium on Formal Methods, LNCS, Springer-Verlag, 2008. This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

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 traveling. 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*. Intuitively, the system invariant is a set S of states from where the system behaves correctly. For example, 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. Obviously, as long the system's state is in S, nothing catastrophic will happen. However, this is not the case when a 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 timer that is responsible for changing sig_1 from red to green is reset due to a circuit problem, sig_1 may turn green within some time while sig_0 is also green. Such a system is called *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 $SPEC_{bt}$, and (2) always meets the following *bounded-time recovery specification* denoted by $SPEC_{br}$: When the system state is in $\neg S$, the system 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 sufficient. In particular, one may require that the system must initially reach a special set of states, say Q, within some time θ, and subsequently recover to S within δ time units. We call the set Q an *intermediate recovery predicate*. The intuition for such *phased recovery* comes from the requirement that the occurrence of faults must be noted (e.g., for scheduling hardware repairs or replacement) *before* normal system operation resumes. Thus, in our example, Q could be the set of states where all signals are red. 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.

In this paper, we concentrate on the problem of synthesizing real-time systems that provide bounded-time phased recovery in the presence of faults. Intuitively, the problem is as follows. After the occurrence of faults, the system must recover to a state in the set Q within θ and from there, recover to the invariant S within δ time units. The main results in this paper are as follows:

- We formally define the notion of bounded-time phased recovery in the context of fault-tolerant real-time systems.
- We show that, in general (i.e., when $Q \not\subseteq S$ and $S \not\subseteq Q$), the problem of synthesizing fault-tolerant real-time programs that provide phased recovery is NP-complete. An example of such a case is the traffic signals system in which Q includes states where all signals are flashing red.
- We characterize a sufficient condition for cases where the synthesis problem can be solved efficiently. In particular, we show that if $S \subseteq Q$, and, execution of the synthesized system needs to be *closed* in Q (i.e., starting from a state in Q, the state of the system never leaves Q) then there exists a polynomial-time sound and complete synthesis algorithm in the size of time-abstract bisimulation of the input intolerant program. An example of such a case is the traffic signals system in which Q is the set of states where either both signals remain red indefinitely or S holds.

Organization of the paper. In Section 2, we formally define real-time programs and the type specifications that we consider in this paper. In Section 3, we present our fault model and introduce the notions of bounded-time phased recovery and fault-tolerance. We formally state the problem of synthesis of fault-tolerant real-time programs that provide bounded-time phased recovery in Section 4. Then, in Section 5, we present our results on the complexity of the synthesis problem and the sufficient condition for existence of a polynomial-time sound and complete synthesis algorithm. In Section 6, we answer potential questions about the techniques presented in this paper and present the related work. Finally, in Section 7, we make concluding remarks and discuss future work.

2 Real-Time Programs and Specifications

In our framework, real-time programs are specified in terms of their state space and their transitions [AH97,AD94]. The definition of specification is adapted from Alpern and Schneider [AS85] and Henzinger [Hen92].

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

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

– *Immediate transitions:* $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$, where s_0 and s_1 are two locations, ν is a clock valuation, and λ is a set of clock variables.
– *Delay transitions:* $(s, \nu) \rightarrow (s, \nu + \delta)$, where s is a location, ν is a clock valuation, and $\delta \in \mathbb{R}_{\geq 0}$ is a *time duration*. We denote a delay transition of duration δ at state σ by (σ, δ).

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

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

Definition 2 (state predicate) A *state predicate* S is any subset of $S_\mathcal{P}$ such that in the corresponding Boolean expression, clock constraints are 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 originates in S then it must remain in S continuously.

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

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

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 $\overline{\sigma}$ is finite and terminates in (σ_f, τ_f) then there does not exist any state σ such that $(\sigma_f, \sigma) \in \psi^s_\mathcal{P}$, 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. *(time consistency)* for all $i \in \mathbb{Z}_{\geq 0}$, (1) if (σ_i, σ_{i+1}) is a delay transition (σ_i, δ) in $\psi^d_\mathcal{P}$ then $\tau_{i+1} - \tau_i = \delta$, and (2) if (σ_i, σ_{i+1}) is an immediate transition in $\psi^s_\mathcal{P}$ then $\tau_i = \tau_{i+1}$. ∎

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

2.2 Example

As mentioned in Section 1, we use the one-lane bridge traffic controller as a running example throughout the 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 program state is updated, and λ is a set of clock variables that are reset by execution of L. Thus, L denotes the set of transitions $\{(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0]) \mid g$ is true in state (s_0, ν), and s_1 is obtained by changing s_0 as prescribed by $st\}$. A *guarded wait command* (also called *delay action*) is of the form $L :: g \longrightarrow$ **wait**, where g identifies the set of states from where delay transitions with arbitrary durations are allowed to be taken as long as g continuously remains true.

The one-lane bridge traffic controller program (TC) has two discrete variables sig_0 and sig_1 with domain $\{G, Y, R\}$ to represent the status of signals. Moreover, for each signal i, $i \in \{0, 1\}$, 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 may turn yellow within 10 time units, but not sooner than 1 time unit. 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. Thus, the traffic controller program is as follows. For $i \in \{0, 1\}$:

```
$TC1_i$ ::  $(sig_i = G) \wedge (1 \leq x_i \leq 10)$   $\xrightarrow{\{y_i\}}$   $(sig_i := Y)$;
[]
$TC2_i$ ::  $(sig_i = Y) \wedge (1 \leq y_i \leq 2)$    $\xrightarrow{\{z_i\}}$   $(sig_i := R)$;
[]
$TC3_i$ ::  $(sig_i = R) \wedge (z_j \leq 1)$          $\xrightarrow{\{x_i\}}$   $(sig_i := G)$;
[]
$TC4_i$ ::  $((sig_i = G) \wedge (x_i \leq 10)) \vee$
            $((sig_i = Y) \wedge (y_i \leq 2)) \vee$
            $((sig_i = R) \wedge (z_j \leq 1))$        $\longrightarrow$         wait;
```

where $j = (i + 1) \mod 2$ and the operator [] denotes non-deterministic choice of execution. Notice that the guard of $TC3_i$ 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.

2.3 Specification

Let $\mathcal{P} = \langle S_\mathcal{P}, \psi_\mathcal{P} \rangle$ be a program. A *specification* (or *property*), denoted *SPEC*, for $\mathcal{P}$ is a set of infinite computations of the form $(\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots$ where $\sigma_i \in S_\mathcal{P}$ for all $i \in \mathbb{Z}_{\geq 0}$. Following Henzinger [Hen92], 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 $\mathcal{P} \models_S \mathit{SPEC}$ 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 $\mathcal{P} \models_S \mathit{SPEC}$ 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 $\mathcal{P}$, there exists a computation suffix $\overline{\beta}$ such that $\overline{\alpha}\overline{\beta} \in \mathit{SPEC}$. We say that $\mathcal{P}$ *violates SPEC* iff it is not the case that $\mathcal{P}$ maintains *SPEC*. ∎

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 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* (see Section 3 for details).

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

The specifications considered in this paper are an intersection of a *safety* specification and a *liveness* specification [AS85,Hen92]. 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. 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. We denote $SPEC_{br}$ 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 $\overline{\alpha} \in \overline{SPEC}$, 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. ■

2.4 Example (cont'd)

Following Definition 8, the safety specification of *TC* comprises of $SPEC_{bt_{TC}}$ and $SPEC_{br_{TC}}$. $SPEC_{bt_{TC}}$ is simply the set of transitions where both signals are not red in their target states:

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

We define $SPEC_{br}$ of *TC* in Section 3, where we formally define the notion of bounded-time phased recovery.

One invariant for the program *TC* is the following:

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

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

3 Fault Model and Fault-Tolerance

3.1 Fault Model

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

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.

Example (cont'd). TC is subject to clock reset faults due to circuit malfunctions. In particular, we consider faults that reset either z_0 or z_1 at any state in the invariant S_{TC} (cf. Subsection 2.3), without changing the location of TC:

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

It is straightforward to see that in the presence of F_0 and F_1, TC may violate $SPEC_{\overline{bt}_{TC}}$. For instance, if F_1 occurs when TC is in a state of S_{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 $TC3_0$ and then $TC3_1$ results in a state where $(sig_0 = sig_1 = G)$, which is clearly a violation of the safety specification. $\blacksquare$

3.2 Phased Recovery and Fault-Tolerance

As illustrated in Section 1, preserving safety specification and providing simple recovery to the invariant from the fault-span may not be sufficient and, hence, it may be necessary to complete recovery to the invariant in a sequence of phases where each phase satisfies certain constraints. We formalize the notion of bounded-time phased recovery by a set of bounded response properties inside the safety specification, i.e., by $SPEC_{br}$ (cf. Definition 8). In this paper, in particular, we focus on 2-*phase recovery*.

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* 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 $SPEC_{\overline{br}}$ from S, where $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \; \wedge \; (Q \mapsto_{\leq \delta} S)$. $\blacksquare$

Note that in Definition 11, if S and Q are disjoint then $\mathcal{P}$ has to recover to Q and then S in order, as S is closed in $\mathcal{P}$. On the other hand, if S and Q are not disjoint, $\mathcal{P}$ has the following options: (1) recover to $Q \cap \neg S$ within θ and then S, or (2) directly recover to $S \cap Q$ within $\min(\delta, \theta)$.

We are now ready to define what it means for a program to be fault-tolerant while providing 2-phase recovery. 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 both absence and presence of faults.

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

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

Example (cont'd). As described in Section 1, when faults F_0 or F_1 (defined in Subsection 3.1) occur, the program TC has to, first, ensure that nothing catastrophic happens and then recover to its normal behavior. Thus, the fault-tolerant version of TC has to, first, reach a state where both signals remain red indefinitely and subsequently recover to S where exactly one signal turns green. In particular, we let the 2-phase recovery specification of TC be the following:

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

where $Q_{TC} = \forall i \in \{0,1\} \ : \ (sig_i = R) \ \wedge \ (z_i > 1)$. The response times in $SPEC_{br_{TC}}$ (i.e., 3 and 7) are simply 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:

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

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

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

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

One may notice that our formulation of the synthesis problem is in spirit close to controller synthesis and game theory problems. However, there exist several differences that are discussed in Section 6.

5 Synthesizing Fault-Tolerant Real-Time Programs with 2-Phase Recovery

5.1 Complexity

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

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

The decision problem (FTPR). 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?

We now show that the FTPR problem is NP-complete to decide. To this end, we reduce a variation of the *2-path problem* [FHW80] to FTPR.

The simplified 2-path problem (2PP). Given are a digraph $G = \langle V, A \rangle$ 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 the vertex v_2 [BJG02].

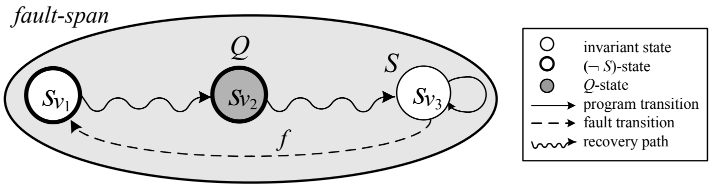

Fig. 1. Mapping 2-path problem to fault-tolerance synthesis.

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

Proof. Since showing membership to NP is straightforward, 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 FTPR 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*) $X = \{\}$,
- (*state space*) $S_\mathcal{P} = \{s_v \mid v \in V\}$,
- (*program transitions*) $\psi_\mathcal{P} = (\{(s_u, s_v) \mid (u, v) \in A\} \cup \{(s_{v_3}, s_{v_3})\}) - \{(s_{v_3}, s_u) \mid u \in V - \{v_3\}\}$,
- (*invariant*) $S = \{s_{v_3}\}$,
- (*fault transitions*) $f = \{(s_{v_3}, s_{v_1})\}$,
- (*safety specification*) $\mathit{SPEC}_{bt} = \{\}$ and $\mathit{SPEC}_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \wedge (Q \mapsto_{\leq \delta} S)$, where $Q = \{s_{v_2}\}$, $\delta = \infty$, and $\theta = \infty$.

An intuitive description of the above mapping is as follows. First, we let the set of clock variables of $\mathcal{P}$ be the empty set. This is simply because our goal is to show that the problem is NP-hard in the size of *locations* of the input program. The state space of $\mathcal{P}$ consists of all vertices in V. The program invariant S merely includes state s_{v_3}. The set of program transitions consists of all arcs in A except arcs that originate at v_3, plus a self-loop at s_{v_3}. Inclusion of the self-loop guarantees that all program computations are infinite. Exclusion of arcs that originate from v_3 ensures the closure of S. There is no bound on recovery times. Finally, we let SPEC_{bt} be the empty set so that all program transitions can potentially participate in the synthesized program.
Reduction. Given the above mapping, we now show that 2PP has a solution iff the answer to the FTPR 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. 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 self-loop (s_{v_3}, s_{v_3}) satisfies the constraints of Problem Statement 1. We prove our claim as follows. Notice that (1) $S_{\mathcal{P}} = S_{\mathcal{P}'}$, (2) $S' = S = \{s_{v_3}\}$, (3) $\psi_{\mathcal{P}'}|S' \subseteq \psi_{\mathcal{P}}|S'$, and (4) $\mathcal{P}'$ is fault-tolerant to *SPEC* from S', as (i) in the absence of faults, by starting from the invariant S', all computations of $\psi_{\mathcal{P}'}$ are infinite, and (ii) in the presence of faults, $\mathcal{P}' \models_{S'} \mathit{SPEC}_{\overline{br}}$ (since Π is a simple path that meets Q and S, respectively), and, $\mathcal{P}' \models_{S'} \mathit{SPEC}_{\overline{bt}}$.

- ($\Leftarrow$) Let the answer to the FTPR problem be $\mathcal{P}' = \langle S_{\mathcal{P}'}, \psi_{\mathcal{P}'} \rangle$ with invariant S'. Since S' must be nonempty, $S' = \{s_{v_3}\}$. Now, consider a computation prefix of $\mathcal{P}'$ that starts from S' and the fault transition (s_{v_3}, 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 $Q \mapsto_{\leq \delta} S'$. Hence, there should exist a computation prefix $\overline{\sigma}$ that originates at $\{s_{v_1}\}$ and reaches $Q = \{s_{v_2}\}$. Moreover, $\overline{\sigma}$ must also visit $S' = \{s_{v_3}\}$. Notice that starting from s_{v_1}, $\overline{\sigma}$ must first visit s_{v_2} and subsequently s_{v_3}. This is because if $\overline{\sigma}$ first visits s_{v_3} then due to the closure of S' in $\psi_{\mathcal{P}'}$, it will never reach s_{v_2}. Now, we claim that that a path, say Π, whose vertices and arcs correspond to state and transitions in $\overline{\sigma}$, is the answer to 2PP. We prove this claim as follows. Obviously, Π starts from v_1, ends at v_3 and contains v_2. Moreover, since $\overline{\sigma}$ reaches both S' and Q in a finite number of steps, $\overline{\sigma}$ cannot contain cycles. Therefore, Π is a simple path. ∎

Example (cont'd). The proof of Theorem 1 particularly implies that if Q and S are disjoint in the problem instance then NP-completeness of the synthesis problem is certain. In the context of *TC*, notice that according to the definitions of S_{TC} and Q_{TC} in Subsections 2.4 and 3.2, it is the case that $S_{TC} \cap Q_{TC} = \{\}$. Hence, the *TC* program and specification in their current form exhibit an instance where the synthesis problem is NP-complete. However, in Subsection 5.3, we demonstrate that a slight modification in the specification of *TC* makes the problem significantly easier to solve. ∎

5.2 A Sufficient Condition for a Polynomial-Time Solution

In this section, we present a sufficient condition under which one can devise a polynomial-time sound and complete solution to the Problem Statement 1 in the size of time-abstract bisimulation of input program.

Claim. Let $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ be a program with invariant S and recovery specification $\mathit{SPEC}_{\overline{br}} \equiv (\neg S \mapsto_{\leq \theta} Q) \ \wedge \ (Q \mapsto_{\leq \delta} S)$. 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 $\psi_{\mathcal{P}'})$. ∎

In order to validate this claim, we propose the Algorithm Add_BoundedPhasedRecovery.

Algorithm sketch. Intuitively, the algorithm works as follows. In Step 1, we transform the input program into a *region graph* [AD94] (described below). In Step 2, we isolate the set of states from where $\mathit{SPEC}_{\overline{bt}}$ may be violated. In Step 3, 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 4, 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 5, we transform the resultant region graph back into a real-time program.

Assumption 1 Let $\overline{\alpha} = (\sigma_0, \tau_0) \rightarrow (\sigma_1, \tau_1) \rightarrow \cdots (\sigma_n, \tau_n)$ 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 $\overline{\alpha}$, only (σ_0, σ_1) is a fault transition and no faults occur outside the program invariant. In our previous work [BK06b], 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_BoundedPhasedRecovery in this paper. Furthermore, notice that the proof of Theorem 1 in its current form holds with this assumption. ∎

Region Graph. Real-time programs can be analyzed with the help of an equivalence relation of finite index on the set of states [AD94]. 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 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 $\tau' < \tau$.

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 $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$ iff for some clock valuations $\nu_0 \in \rho_0$ and $\nu_1 \in \rho_1$, $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$ is a transitions in $\psi_\mathcal{P}$.

We now describe the algorithm Add_BoundedPhasedRecovery 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 $U^r = \{(s, \rho) \mid \exists (s, \nu) : ((s, \nu) \in U \ \wedge \ \nu \in \rho)\}$.

Algorithm 1 Add_BoundedPhasedRecovery

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

- (*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 $T_1^r - Q^r$, Q^r, and θ to ensure that $R(\mathcal{P})$ indeed satisfies the bounded response property $\neg S \mapsto_{\leq \theta} Q$. The details of how the procedure Add_BoundedResponse (first proposed in [BK06a]) functions are not provided in this paper, with the exception of the following properties: (1) it adds a clock variable, say t_1, which gets reset when $T_1 - Q$ becomes true, to the set X of clock variables of $\mathcal{P}$, (2) 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 (3) 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_1^r. Likewise, in Line 10, the algorithm adds a clock variable, say t_2, which gets reset when $Q - S_1$ becomes true and ensures that $R(\mathcal{P})$ satisfies $Q \mapsto_{\leq \delta} S_1$.

- (*Step 4*) Since we remove the set ns of regions from T_1^r, 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 $\langle S_{\mathcal{P}}^r, \psi_{\mathcal{P}_1}^r \rangle$ does not have deadlock regions in the region invariant S_1^r (Lines 15-17). Precisely, we say that a region (s_0, ρ_0) of region graph $R(\mathcal{P}) = \langle S_{\mathcal{P}}^r, \psi_{\mathcal{P}}^r \rangle$ 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 $(s_0, \rho_0) \rightarrow (s_1, \rho_1) \in \psi_{\mathcal{P}}^r$. 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 5*) Finally, upon reaching a fixpoint, we transform the resulting region graph $\langle S_{\mathcal{P}}^r, \psi_{\mathcal{P}_1}^r \rangle$ 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* [AD94], we will not lose any behaviors in the reverse transformation.

We now show that the algorithm Add_BoundedPhasedRecovery 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 2. *The Algorithm Add_BoundedPhasedRecovery is sound.*

Proof. We show that the algorithm satisfies the constraints of Problem Statement 1. Let the algorithm add two clock variables t_1 and t_2 when invoking Add_BoundedResponse (cf. Lines 8 and 10). We proceed as follows:

1. (*Constraints* $C1..C3$) By construction, correctness of these constraints trivially follows.
2. (*Constraint* $C4$) We distinguish two subgoals based on the behavior of $\mathcal{P}'$ in the absence and presence of faults:
   - We need to show that in the absence of faults, $\mathcal{P}' \models_{S'} \mathit{SPEC}$. 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 $\overline{\sigma} \in \mathit{SPEC}$. 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, $\mathcal{P}' \models_{S'} \mathit{SPEC}$.
   - 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 1, none of the subsequent transitions in $\overline{\sigma}$ is 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 3. *The Algorithm Add_BoundedPhasedRecovery is complete.*

Proof sketch. 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., not reached). 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

$C3$ of the Problem Statement 1, any transition that begins in the invariant must be a transition of the fault-intolerant program. Due to closure 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 the 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).

Our algorithm declares failure when either the invariant or fault-span of the synthesized program is equal to the empty set. In other words, our algorithm fails to find a solution when all states of the intolerant program are illegitimate with respect to Problem Statement 1. Therefore, the algorithm Add_BoundedPhasedRecovery is complete. ∎

5.3 Example (cont'd)

We now demonstrate how the algorithm Add_BoundedPhasedRecovery synthesizes a fault-tolerant version of TC, which provides bounded-time recovery. In the recovery specification of TC (cf. in Subsection 3.2), the invariant predicate S_{TC} and intermediate recovery predicate Q_{TC} were disjoint and in Subsection 5.1, we showed such specifications make the synthesis problem NP-complete. Now, let the intermediate recovery predicate be:

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

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_{TC} within 3 time units. Since, $S_{TC} \subseteq Q_{new}$, we apply the Algorithm Add_BoundedPhasedRecovery to transform TC into a fault-tolerant program TC'. We note that due to many symmetries in TC and the complex structure of the algorithm, we only present a highlight of the process of synthesizing TC'.
First, observe that in Step 2 of the algorithm, $ms = \{\}$ and $mt = SPEC_{bt_{TC}}$. 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:

$$TC5_i \ :: \quad (sig_0 = sig_1 = R) \ \wedge \ (z_0, z_1 \leq 2) \ \wedge \ (t_1 \leq 2) \quad \longrightarrow \quad \textbf{wait};$$

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 $(sig_0 = sig_1 = R) \wedge (z_0, z_1 > 1)$.
Comment. 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 $TC5$ lets the program wait only for 2 time units. This is because Add_BoundedResponse first includes

computations with the smallest possible time delay and *optionally* includes additional computations to increase the level non-determinism. In this context, other such additional computations may be constructed by the following actions for sig_0:

$$\begin{array}{lllll}
TC6_0 :: & (sig_0 = sig_1 = R) \ \wedge \ (z_0, z_1 \leq 1) \ \wedge \ (t_1 \leq 1) & \xrightarrow{\{x_0\}} & (sig_0 \ := \ G); \\
TC7_0 :: & (sig_0 = G) \wedge (sig_1 = R) \wedge & & \\
 & (x_0 = 1) \ \wedge \ (z_0, z_1 \leq 1) \ \wedge \ (t_1 \leq 2) & \xrightarrow{\{y_0\}} & (sig_0 \ := \ Y); \\
TC8_0 :: & (sig_0 = G) \wedge (sig_1 = R) \wedge & & \\
 & (x_0 \leq 1) \ \wedge \ (z_0, z_1 \leq 1) \ \wedge \ (t_1 \leq 2) & \longrightarrow & \textbf{wait};
\end{array}$$

Notice that while executing recovery action $TC6_0$ results in reaching another state in $T_1 - Q_{new}$, execution of actions $TC7_0$ and $TC8_0$ result in reaching a state in invariant S_{TC}, which is clearly in Q_{new} as well. ∎

Now, consider the case where TC is in a state where $(sig_0 = G) \wedge (sig_1 = R) \ \wedge \ (x_0 = 1) \ \wedge \ (z_0, z_1 \leq 1)$. In this case, one may argue that TC has the option of executing action $TC3_1$ and reaching a state where $sig_0 = sig_1 = G$, which is clearly a violation of safety specification $SPEC_{\overline{bt}_{TC}}$. However, since we remove the set mt from $\psi_{\mathcal{P}_1}$ (Line 7), action $TC3_i$ would be revised as follows:

$$TC3_i:: \quad (sig_i = R) \ \wedge \ (z_j \leq 1) \ \wedge \ (sig_j \neq G) \quad \xrightarrow{\{x_i\}} \quad (sig_i := G);$$

for all $i \in \{0, 1\}$ where $j = (i+1) \mod 2$. In other words, the algorithm strengthens the guard of $TC1_i$ such that in the presence of faults, a signal does not turn green while the other one is also green.

In Step 4, consider the state predicate $Q_{new} - S1_{TC} = (sig_0 = sig_1 = R) \ \wedge \ (z_0, z_1 > 1)$. Similar to Step 3, the algorithm adds recovery paths with the smallest possible time delay, which is the following action for either $i = 0$ or $i = 1$:

$$TC9_i:: \quad (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \quad \xrightarrow{\{z_i\}} \quad \textbf{skip};$$

It is straightforward to verify that by execution of $TC9_i$, the program reaches the invariant S_{TC} from where the program behaves correctly. Similar to Step 3, the procedure Add_BoundedResponse may include the following additional actions:

$$\begin{array}{llll}
TC10_i:: & (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \ \wedge \ (t_2 \leq 7) & \xrightarrow{\{x_i\}} & (sig_i := G); \\
TC11_i:: & (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \ \wedge \ (t_2 \leq 7) & \xrightarrow{\{y_i\}} & (sig_i := Y); \\
TC12_i:: & (sig_i = sig_j = R) \ \wedge \ (z_i, z_j > 1) \ \wedge \ (t_2 \leq 7) & \longrightarrow & \textbf{wait};
\end{array}$$

Comment. One may notice that action $TC11_i$ adds a strange behavior to 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 $SPEC_{\overline{bt}_{TC}}$. 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 $SPEC_{bt_{TC}}$. 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 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 Discussion

The impact of the NP-completeness result and the polynomial-time algorithm. Let S be the set of legitimate states of the system (invariant) in the absence of faults and Q be the set of states where recovery must occur before legitimate operation is continued. Observe that the NP-completeness result in Subsection 5.1 can be trivially extended to the case where $Q \cap S \neq \{\}$, although $Q \not\subseteq S$ and $S \not\subseteq Q$. This can be seen from a simple reduction from the case where $Q \cap S = \{\}$ and then adding a new state, say x, to both S and Q but requiring that any transitions that reach are in $SPEC_{bt}$, i.e., they violate safety. Also, the NP-completeness is not caused due to problem formulation about timing constraints. In particular, even if we were to require constraints of the form $SPEC_{\overline{br}} \equiv (\neg S \mapsto_{\leq \delta} S) \wedge (Q \mapsto_{\leq \theta} S)$, the proof in Subsection 5.1 is still valid. When faced with the problem where the phased recovery problem is NP-complete, the designer may be able to change the problem at hand slightly to make it fall into the category where polynomial-time synthesis is feasible. For example, given an instance of the phased recovery problem with predicates S and Q, the designer could instead consider the problem where $Q_{new} = S \cup Q$. In case of the one-lane bridge problem, we exploited this observation without introducing undesirable recovery computations. However, this is not always the case and the synthesized program may have certain undesirable properties (that may have to be fixed later). Nevertheless, even if the modified specification is not acceptable, there is a potential to use *partial automation* where we first synthesize the program with modified specification and then subsequently modify it manually.

Automated synthesis of hard-masking programs. In [BK06b], we introduced the notion of *hard-masking* fault-tolerance in which a program must satisfy its safety and liveness properties as well its timing-constraints in both absence and presence of faults and provides simple bounded-time recovery. We showed that the problem of synthesizing hard-masking programs that satisfy two bounded response properties and provide simple recovery is NP-complete. However, we conjectured that the complexity of the synthesis problem where the program should satisfy one timing constraint and provide simple recovery is exponential. The fault-tolerant programs we considered in this paper fall in the category of hard-masking programs with one timing constraint $Q \mapsto_{\leq \delta} S$ and recovery constraint $\neg S \mapsto_{\leq \theta} S$. Hence, the NP-completeness result in Subsection 5.1 validates our conjecture on the complexity of the problem of synthesizing hard-masking programs.

Related Work. Our formulation of the synthesis problem is in spirit close to timed controller synthesis (e.g., [BDMP03,DM02,AM99,AMPS98]), where program and fault transitions may be modeled as controllable and uncontrollable actions, and game theory (e.g., [dAFH$^+$03,FLM02]), where program and fault transitions may be modeled in terms of two players. In controller synthesis (respectively, game theory) the objective is to *restrict* the actions of a *plant* (respectively, an *adversary*) at each state through synthesizing a *controller* (respectively, a *wining strategy*) such that the behavior of the entire system always meets some safety and/or reachability conditions. Notice that the conditions $C1..C3$ in Problem Statement 1 precisely express this notion of restriction (also called *language inclusion*). Moreover, constraint $C4$ implicitly implies that the synthesized program is not allowed to exhibit new finite computations, which is known as the *non-blocking* condition. Note, however, that there are several distinctions. First, in addition to safety and reachability constraints, our notion of fault-tolerance is also concerned with adding new *bounded-time recovery* behaviors to the given program as well, which is normally not a concern in controller synthesis and game theory. Secondly, unlike most game theoretic approaches, we do not consider turns between occurrence of program and fault transitions. Thirdly, in controller synthesis and game theory, a common assumption is that the existing program and/or the given specification must be deterministic which is not the case in our model. Finally, we concentrate on safety properties typically used in specifying real-time systems (cf. Definition 8). As a result, the complexity of our synthesis techniques is often lower than the related work. For example, synthesis problems presented in [dAFH$^+$03,FLM02,AMPS98,AM99] are EXPTIME-complete and deciding the existence of a controller in [DM02,BDMP03] is 2EXPTIME-complete.

7 Conclusion and Future Work

In this paper, we focused on the problem of synthesizing fault-tolerant real-time programs that mask the occurrence of faults while providing bounded-time phased recovery. We modeled such phased recovery using bounded response properties of the form $(\neg S \mapsto_{\leq \theta} Q) \wedge (Q \mapsto_{\leq \delta} S)$ where S is an invariant predicate and Q is an intermediate recovery predicate. We showed that in general the problem is NP-complete in the size of locations of the input program. We also showed that if $S \subseteq Q$ and Q is closed in execution of the output program then there exists a polynomial-time solution to the problem in the size of the input program's region graph.

We note that the complexity class remains the same even if the phased recovery problem is extended to the case where more than 2 phases (e.g., recovery to Q_1, recovery to Q_2 and recovery to invariant S) are concerned. In particular, if $Q_1 \supseteq Q_2 \supseteq S$ and Q_1 and Q_2 are closed in the program to be synthesized then the phased recovery can be added in polynomial-time in the region graph.

Also, as discussed in Subsection 5.3, the designer can use the contrast between the complexity classes with slightly different problem specifications to determine if system requirements can be slightly modified for permitting automated synthesis. In particular, in Section 6, we argued that the alternate specification (where the problem is in P) for the one-lane bridge problem considered in this paper may be acceptable to many designers. Also, as argued in that section, the modified specification can assist in partial automation of providing fault-tolerance with phased recovery. One of our future works in this area is to develop algorithms that utilize such a partial automation.

We are currently working on other variations of the problem. One such variation is where $S \subseteq Q$, but Q need *not* be closed in the output program. We conjecture that the complexity of this problem is exponential. We also plan to develop symbolic algorithms for synthesizing bounded-time phased recovery. In previous work, we have shown that such techniques are extremely effective in synthesizing distributed programs with state space of size 10^{30} and beyond [BK07].

References

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AH97] 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.

[AM99] 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.

[AMPS98] 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.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[BDMP03] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.

[BJG02] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2002.

[BK06a] 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.

[BK06b] 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.

[BK07] 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.

[dAFH$^+$03] 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.

[DM02] 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.

[FHW80] S. Fortune, J. E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[FLM02] M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.

[Hen92] T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.