Incremental Synthesis of Fault-Tolerant Real-Time Programs[1]

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
http://www.cse.msu.edu/~{borzoo,sandeep}

Abstract. In this paper, we focus on the problem of automated addition of fault-tolerance to an existing fault-intolerant *real-time* program. We consider three levels of fault-tolerance, namely *nonmasking*, *failsafe*, and *masking*, based on safety and liveness properties satisfied in the presence of faults. More specifically, a nonmasking (respectively, failsafe, masking) program satisfies liveness (respectively, safety, both safety and liveness) in the presence of faults. For failsafe and masking fault-tolerance, we consider two additional levels, *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. We present a polynomial time algorithm (in the size of the input program's region graph) that adds *bounded-time recovery* from an arbitrary given set of states to another arbitrary set of states. Using this algorithm, we propose a sound and complete synthesis algorithm that transforms a fault-intolerant real-time program into a nonmasking fault-tolerant program. Furthermore, we introduce sound and complete algorithms for adding soft/hard-failsafe fault-tolerance. For reasons of space, our results on addition of soft/hard-masking fault-tolerance are presented in a technical report.

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

1 Introduction

Automated program synthesis is the problem of designing an algorithmic method to find a program that satisfies a required set of properties. Such automated synthesis is desirable, as it ensures that the synthesized program is correct-by-construction. In existing specification-based synthesis methods, a change in the specification requires us to redo synthesis from scratch. Thus, it would be advantageous, if we could reuse the previous

[1] This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

efforts made to synthesize real-time programs and somehow *incrementally add* properties (e.g., fault-tolerance) to them. Moreover, such incremental synthesis is especially useful if the given real-time program is designed manually, e.g., for ensuring that the original program is efficient. More importantly, incremental synthesis is crucial when the existing real-time program satisfies properties whose automated synthesis is undecidable (e.g., precise eventuality $\Diamond_{=\delta} q$) or lies in highly complex classes of complexity.

In this paper, we focus on designing incremental synthesis algorithms that solely add fault-tolerance to existing fault-intolerant real-time programs, where processes can read and write all program variables in one atomic step. In particular, we concentrate on algorithms with manageable time and space complexity such that they can be used in tools for synthesizing fault-tolerant real-time programs. To characterize such manageable complexity, we require that the complexity of our algorithms are comparable to that of existing model checking techniques in the dense real-time model.

In order to characterize fault-tolerance requirements of programs, in our work, we consider three levels of fault-tolerance, namely *nonmasking* (respectively, stabilizing), *failsafe*, and *masking*, based on safety and liveness properties satisfied in the presence of faults. Furthermore, we propose two additional levels, namely *soft* and *hard* fault-tolerance, based on satisfaction of timing constraints in the presence of faults. Precisely, in the absence of faults, both soft and hard fault-tolerant programs are required to satisfy their timing constraints. However, in the presence faults, a soft fault-tolerant program is *not* required to satisfy its timing constraints while a hard fault-tolerant program is required to do so. In this sense, for instance, a hard-failsafe program satisfies its safety specification as well as its timing constraints in the presence of faults.

1.1 Related Work

In the literature of real-time computing, fault-tolerance has mostly been addressed in the context of scheduling theory (e.g., [1, 2]). In fault-tolerant real-time scheduling, the objective is to find the optimal schedule of a set of tasks on a set of processors *dynamically*, such that the largest possible number of tasks meet their deadlines. Since time complexity is a critical issue in dynamic scheduling, most of the proposed algorithms are in the form of heuristics designed for specific platforms and special types of faults (e.g., transient, fail-stop, Byzantine, etc.).

Recently, we studied the problem of incremental synthesis of timed automata in the absence of faults in [3]. More specifically, we developed synthesis algorithms and hardness results for adding different types of bounded response properties to a given timed automaton. We also studied the problem of incremental addition of UNITY [4] properties to untimed programs in [5].

The problem of synthesizing *untimed* fault-tolerant programs has been studied in the literature from different perspectives. In [6–8], the authors propose synthesis methods for adding fault-tolerance and multitolerance to existing untimed programs. In [9], Attie, Arora, and Emerson study the problem of synthesizing fault-tolerant concurrent untimed programs from temporal logic specifications expressed in CTL formulas.

Synthesis of real-time systems has mostly been studied in the context of controller synthesis and game theory [10–15]. In these papers, the common assumption is that the existing program (called a plant) and/or the given specification are *deterministic*.

Moreover, since the authors consider highly expressive specifications, the complexity of proposed methods is very high. For example, synthesis problems presented in [10,11, 14, 15] are EXPTIME-complete. Moreover, deciding the existence of a solution (called a controller) in [12, 13] is 2EXPTIME-complete.

1.2 Contributions

In this paper we (i) introduce a generic fault-tolerance framework for real-time programs independent of platform, architecture, and type of faults; (ii) extend the previous work by Kulkarni and Arora [6] to the context of real-time programs; (iii) consider a general notion of real-time programs that covers both deterministic and nondeterministic programs in both synchronous and asynchronous models; and (iv) introduce various levels of fault-tolerance for real-time systems based on satisfaction of properties and timing constraints in the presence of faults. Furthermore, we present a class of specifications where we can express typical requirements for specifying real-time and fault-tolerant systems and we show that the complexity of synthesis algorithms for this class of specifications is comparable to existing model checking techniques for real-time programs [16]. Moreover, since we follow the standard model of timed automata [17], many of the problems in fault-tolerant scheduling theory can be modeled in our framework [18].

The main results in this paper are as follows. First, we present a polynomial time algorithm (in the size of the input program's region graph) that adds bounded-time recovery from an arbitrary given set of states to another arbitrary set of states. Then, using this algorithm, we propose sound and complete synthesis algorithms that transform a fault-intolerant real-time program into a (1) nonmasking or soft-failsafe fault-tolerant programs, or (2) hard-failsafe fault-tolerant program where the synthesized fault-tolerant program is required to satisfy at most one bounded response property in the presence of faults. For reasons of space, in a technical report [19], we also present a synthesis algorithm for adding soft-masking fault-tolerance. Moreover, we show that the problem of adding hard-masking fault-tolerance where the synthesized program is required to satisfy at least two bounded response properties in the presence of faults is NP-hard.

Organization of the paper. In Section 2, we present formal definitions of real-time programs, specifications, and regions graphs. We introduce the notions of faults and fault-tolerance in the context of real-time programs in Section 3. In Section 4, we formally state the problem of adding fault-tolerance to real-time programs. We present our synthesis algorithms for adding nonmasking, soft-failsafe, and hard-failsafe fault-tolerance in sections 5, 6, and 7, respectively. Finally, in Section 8, we make the concluding remarks.

2 Real-Time Programs, Specifications, and Region Graphs

In our framework, programs are specified in terms of their state space and their transitions [20]. The definition of specifications is adapted from Henzinger [21]. Finally, the notion of region graph is due to Alur and Dill [17].

2.1 Real-Time Program

Let V be a finite set of *discrete variables* and X be a finite set of *clock variables*. Each discrete variable is associated with a finite *domain* D of values. A *location* is a function that maps each discrete variable to a value from its respective domain. 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 $\Phi(X)$. A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. Furthermore, for $\tau \in \mathbb{R}_{\geq 0}$, $\nu + \tau = \nu(x) + \tau$ for every clock x. Also, for $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation for X which 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), such that s is a location and ν is a clock valuation for X at location s. A *transition* (denoted (σ_0, σ_1)) is of the form $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$. Transitions are classified into two types:

- **Delay:** for a state $\sigma = (s, \nu)$ and a time *duration* $\delta \in \mathbb{R}_{\geq 0}$ (denoted (σ, δ)), $(s, \nu) \rightarrow (s, \nu + \delta)$.
- **Jump:** for a state (s_0, ν), a location s_1, and a set λ of clock variables, $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$.

A *program* $\mathcal{P}$ is a tuple $\langle S_p, \psi_p \rangle$, where S_p is the *state space*, and ψ_p is a set of transitions. Let ψ_p^s and ψ_p^d denote the set of jump and delay transitions in ψ_p, respectively. A *state predicate* is a subset of S_p such that it is definable by the above syntax of clock constraints, i.e., in the corresponding Boolean expression clock variables are only compared to nonnegative integers. A state predicate S is *closed* in program $\mathcal{P}$ if $((\forall (\sigma_0, \sigma_1) \in \psi_p^s \ : \ (\sigma_0 \in S \Rightarrow \sigma_1 \in S)) \ \wedge \ (\forall (\sigma, \delta) \in \psi_p^d \ : \ (\sigma \in S \Rightarrow \forall \epsilon \leq \delta \ : \ \sigma + \epsilon \in S)))$. A timed state sequence $\langle (\sigma_0, \tau_0), (\sigma_1, \tau_1), \cdots \rangle$, where $\tau_i \in \mathbb{R}_{\geq 0}$, is a *computation* of $\mathcal{P}$ if the following conditions are satisfied: (1) $\forall j > 0 : (\sigma_{j-1}, \sigma_j) \in \psi_p$, (2) if it is finite and terminates in (σ_l, τ_l) then there does not exist state σ such that $(\sigma_l, \sigma) \in \psi_p$, and (3) the sequence $\langle \tau_0, \tau_1, \cdots \rangle$ satisfies the following constraints:

Monotonicity: $\tau_i \leq \tau_{i+1}$ for all $i \in \mathbb{N}$.
Divergence: For all $t \in \mathbb{R}_{\geq 0}$, there exists j such that $\tau_j \geq t$.

The *projection* of a set of program transitions ψ_p on state predicate S (denoted $\psi_p|S$) is the set of transitions $\{(\sigma_0, \sigma_1) \in \psi_p^s \mid \sigma_0, \sigma_1 \in S\} \cup \{(\sigma, \delta) \in \psi_p^d \mid \sigma \in S \ \wedge \ (\forall \epsilon \leq \delta \ : \ \sigma + \epsilon \in S)\}$.

2.2 Specification

A *specification* (or *property*), denoted Σ, is a set of timed state sequences of the form $\langle (\sigma_0, \tau_0), (\sigma_1, \tau_1), \cdots \rangle$. Following Henzinger [21], we require that the sequence $\langle \tau_0, \tau_1, \cdots \rangle$ satisfies monotonicity and divergence. We now define what it means for a program $\mathcal{P}$ to satisfy a specification Σ. Given a program $\mathcal{P}$, a state predicate S, and a specification Σ, we write $\mathcal{P} \models_S \Sigma$ and say that program $\mathcal{P}$ *satisfies* Σ *from* S iff (1)

S is closed in $\mathcal{P}$, and (2) every computation of $\mathcal{P}$ that starts where S is true is in Σ. If $\mathcal{P} \models_S \Sigma$ and $S \neq \{\}$, we say that S is an *invariant* of $\mathcal{P}$ for Σ.
Notation. 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 Σ".

We say that program $\mathcal{P}$ *maintains* Σ iff for all finite timed state sequences α of $\mathcal{P}$, there exists a timed state sequence β such that $\alpha\beta \in \Sigma$. We say that $\mathcal{P}$ *violates* Σ iff it is not the case that $\mathcal{P}$ maintains Σ. Note that, the definition of *maintains* identifies the property of finite timed state sequences, whereas the definition of *satisfies* expresses the property of infinite timed state sequences.

Following Alpern and Schneider [22] and Henzinger [21], we let the specification consist of a *liveness specification* and a *safety specification*. The liveness specification is represented by a set of infinite computations. A program satisfies the liveness specification, if every computation prefix of the program has a suffix that is in the liveness specification.

Remark 2.1: In the synthesis problem, we begin with an initial fault-intolerant program that satisfies its specification (including the liveness specification) in the absence of faults. We will show that our synthesis algorithms *preserve* liveness specification. Hence, the liveness specification need not be specified explicitly.

In this paper, with abuse of notation, we let the safety specification consist of (1) a set Σ_{bt} of *bad transitions* that should not occur in the program computation, i.e., a subset of $\{(\sigma_0, \sigma_1) \mid \sigma_0, \sigma_1 \in S_p\}$, and (2) a conjunction of zero or more *bounded response* properties of the form $\Sigma_{br} \equiv ((P_1 \mapsto_{\leq \delta_1} Q_1) \wedge (P_2 \mapsto_{\leq \delta_2} Q_2) \ \wedge\ ... \wedge (P_m \mapsto_{\leq \delta_m} Q_m))$, i.e., it is always the case that a state in P_i is followed by a state in Q_i within δ_i time units, where P_i and Q_i are state predicates and $\delta_i \in \mathbb{Z}_{\geq 0}$, for all i such that $1 \leq i \leq m$. Observe that we abuse the $\models$ notation for the set Σ_{bt} of bad transitions. This is because it is possible to trivially translate this concise representation of safety into the corresponding set of infinite computations. The same concept applies to definitions of *maintains* and *violates*.

2.3 Region Graph

Real-time programs can be analyzed with the help of an equivalence relation of finite index on the set of states [17]. Given a real-time program $\mathcal{P}$, for each clock $x \in X$, let c_x be the largest constant in the guards of transitions and invariant of $\mathcal{P}$ that involve x, where $c_x = 0$ if x does not occur in any guard or invariant of $\mathcal{P}$. 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 (3) for all $x \in \{y \in X \mid \nu(y) < c_y\}$, the clock value $\nu(x)$ is an integer if and only if $\mu(x)$ 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* is an equivalence class with respect to $\equiv$. Using the region equivalence relation, we construct the *region graph* of $\mathcal{P}\langle S_p, \psi_p\rangle$ (denoted $R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$) as follows. Vertices of $R(\mathcal{P})$ (denoted S_p^r) are regions. Edges of $R(\mathcal{P})$ (denoted ψ_p^r) 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 ψ_p. We say that a region (s_0, ρ_0) of region graph

$R(\mathcal{P})$ is a *deadlock region* iff for all regions (s_1, ρ_1), there does not exist an edge of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$. A *region predicate* S^r with respect to a state predicate S is defined by $S^r = \{(s, \rho) \mid \exists (s, \nu) : ((s, \nu) \in S \ \wedge \ \nu \in \rho)\}$. Likewise, the region predicate with respect to invariant S of a program $\mathcal{P}$ is called *region invariant* S^r. The projection of a set of edges ψ_p^r on region predicate S^r (denoted $\psi_p^r | S^r$) is the set of edges $\{(r_0, r_1) \in \psi_p^r \mid r_0, r_1 \in S^r\}$.

Region graphs are time-abstract bisimulation of real-time programs [17]. In our synthesis algorithms in section 5, 6, and 7, we transform a real-time program $\mathcal{P}\langle S_p, \psi_p \rangle$ into its corresponding region graph $R(\mathcal{P})\langle S_p^r, \psi_p^r \rangle$ by invoking the procedure ConstructRegionGraph. We also let this procedure take state predicates and sets of transitions in $\mathcal{P}$ (e.g., S and Σ_{bt}) and return the corresponding region predicates and sets of edges in $R(\mathcal{P})$ (e.g., S^r and Σ_{bt}^r). Likewise, we transform a region graph $R(\mathcal{P})$ back to a real-time program by invoking the procedure ConstructRealTimeProgram.

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 $\tau' < \tau$. We call a region (s, ρ) a *boundary region*, if for each $\nu \in \rho$ and for any $\tau \in \mathbb{R}_{\geq 0}$, ν and $\nu + \tau$ are not equivalent. A region is *open*, if it is not a boundary region. A region (s, ρ) is called an *end region*, if $\nu(x) > c_x$ for all $\nu \in \rho$ and for all clocks $x \in X$.

3 Faults and Fault-Tolerance in Real-Time Programs

In this section, we extend formal definitions of faults and fault-tolerance due to Arora and Gouda [23] to the context of real-time programs. The faults that a program is subject to are systematically represented by transitions. A class of *faults* f for program $\mathcal{P}\langle S_p, \psi_p \rangle$ is a subset of the set $S_p \times S_p$. Faults are also categorized into *delay faults* and *jump faults*. We use $\psi_p[]f$ to denote the transitions obtained by taking the union of the transitions in ψ_p and the transitions in f.

We say that a state predicate T is an f-span (read as *fault-span*) of $\mathcal{P}$ from S iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) T is closed in $\psi_p[]f$. Observe that for all computations of $\mathcal{P}$ 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. As we defined the computations of $\mathcal{P}$, we say that a timed state sequence, $\langle (\sigma_0, \tau_0), (\sigma_1, \tau_1), \cdots \rangle$, is a *computation of* $\mathcal{P}$ *in the presence of* f iff the following four conditions are satisfied: (1) $\forall j > 0 : (\sigma_{j-1}, \sigma_j) \in (\psi_p \cup f)$, (2) if it is finite and terminates in state (σ_l, τ_l) then there does not exist state σ such that $(\sigma_l, \sigma) \in \psi_p$, (3) $\langle \tau_0, \tau_1, \cdots \rangle$ satisfies monotonicity and divergence, and (4) $\exists n \geq 0 : (\forall j > n : (\sigma_{j-1}, \sigma_j) \in \psi_p)$.

We consider three levels of fault-tolerance, namely nonmasking, failsafe, and masking based on satisfaction of safety and liveness properties in the presence of faults. For failsafe and masking fault-tolerance, we propose two additional levels, namely *soft* and *hard*, based on satisfaction of timing constraints in the presence of faults. Intuitively, a *soft fault-tolerant* real-time program is not required to satisfy its timing constraints in the presence of faults. A *hard fault-tolerant* real-time program must satisfy its timing constraints even in the presence of faults.

Let specification Σ consist of Σ_{bt} and Σ_{br}. Since a nonmasking fault-tolerant program need not satisfy safety in the presence of faults, $\mathcal{P}$ is *nonmasking f-tolerant from S for Σ with recovery time δ*, where $\delta \in \mathbb{Z}_{\geq 0}$, iff (1) $\mathcal{P} \models_S \Sigma_{bt}$, (2) $\mathcal{P} \models_S \Sigma_{br}$, and (3) there exists T such that T is an f-span of $\mathcal{P}$ from S, and every computation of $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ that starts from a state in T, reaches a state in S within δ time units. We say that $\mathcal{P}$ is *soft-failsafe f-tolerant from S for Σ* iff (1) $\mathcal{P} \models_S \Sigma_{bt}$, (2) $\mathcal{P} \models_S \Sigma_{br}$, and (3) there exists T such that T is an f-span of $\mathcal{P}$ from S, and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{bt} from T. A program $\mathcal{P}$ is *hard-failsafe f-tolerant from S for Σ* iff $\mathcal{P}$ is soft-failsafe f-tolerant from S for Σ and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{br} from T. A program $\mathcal{P}\langle S_p, \psi_p\rangle$ is *soft-masking f-tolerant from S for Σ with recovery time δ*, where $\delta \in \mathbb{Z}_{\geq 0}$, iff (1) $\mathcal{P} \models_S \Sigma_{bt}$, (2) $\mathcal{P} \models_S \Sigma_{br}$, (3) there exists T such that T is an f-span of $\mathcal{P}$ from S and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{bt} from T, and (4) every computation of $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ that starts from a state in T, reaches a state in S within δ time units. A program $\mathcal{P}\langle S_p, \psi_p\rangle$ is *hard-masking f-tolerant from S for Σ with recovery time δ*, where $\delta \in \mathbb{Z}_{\geq 0}$, iff $\mathcal{P}$ is soft-masking f-tolerant from S for Σ with recovery time δ, and $\mathcal{P}\langle S_p, \psi_p[]f\rangle$ maintains Σ_{br} from T.
Notation. Whenever the specification Σ and the invariant S are clear from the context, we omit them; thus, "f-tolerant" abbreviates "f-tolerant from S for Σ".
Assumption 3.1: Since $\mathcal{P}$ satisfies $\Sigma_{br} \equiv ((P_1 \mapsto_{\leq \delta_1} Q_1) \wedge \ldots \wedge (P_m \mapsto_{\leq \delta_m} Q_m))$ in the absence of faults (cf. Remark 2.1), without loss of generality, we assume that for each bounded response property $(P_i \mapsto_{\leq \delta_i} Q_i)$, where $1 \leq i \leq m$, the intolerant program already has a clock variable that is reset on transitions that go from a state in $\neg P_i$ to a state in P_i to keep track of time as soon as P_i becomes true.
Assumption 3.2: We assume that faults are immediately detectable and that given a state of the program, we can determine the number of faults that have occurred in reaching that state. This assumption is needed only for addition of hard fault-tolerance and is realistic in many commonly considered systems. For instance, in multiprocessor scheduling theory, a processor-crash is immediately detectable and its number of occurrences is easily traceable.
Assumption 3.3: We assume that the number of occurrence of faults in a program computation is bounded by a pre-specified value n. This assumption is required since for commonly considered faults, it can be shown that *bounded-time recovery* in the presence of unbounded occurrence of faults is impossible.

4 Problem Statement

Given are a fault-intolerant real-time program $\mathcal{P}\langle S_p, \psi_p\rangle$, its invariant S, a set of faults f, and a safety specification Σ such that $\mathcal{P} \models_S \Sigma$. Our goal is to synthesize a real-time program $\mathcal{P}'\langle S_p, \psi'_p\rangle$ with invariant S' such that $\mathcal{P}'$ is f-tolerant from S' for Σ. As mentioned in the introduction, our synthesis methods obtain $\mathcal{P}'$ from $\mathcal{P}$ by adding fault-tolerance *alone* to $\mathcal{P}$, i.e., $\mathcal{P}'$ does not introduce new behaviors to $\mathcal{P}$ when no faults have occurred. Observe that:

1. If S' contains states that are not in S then, in the absence of faults, $\mathcal{P}'$ may include computations that start outside S. Since we require that $\mathcal{P}' \models_{S'} \Sigma$, it would imply that $\mathcal{P}'$ is using a new way to satisfy Σ in the absence of faults.

2. If $\psi'_p|S'$ contains a transition that is not in $\psi_p|S'$ then $\mathcal{P}'$ can use this transition in order to satisfy Σ in the absence of faults.

Thus, the synthesis problem is as follows (we instantiate this problem for soft/hard-failsafe, nonmasking, and soft/hard-masking f-tolerance in the obvious way):
Problem Statement 4.1. Given $\mathcal{P}\langle S_p, \psi_p\rangle$, S, Σ, and f such that $\mathcal{P} \models_S \Sigma$.
Identify $\mathcal{P}'\langle S_p, \psi'_p\rangle$ and S' such that

$(C1)$ $S' \subseteq S$
$(C2)$ $\psi'_p|S' \subseteq \psi_p|S'$, and
$(C3)$ $\mathcal{P}'$ is f-tolerant from S' for Σ.

Soundness and completeness. We say that an algorithm for the synthesis problem is *sound* iff its output meets the constraints of the Problem Statement 4.1. We say that an algorithm for the synthesis problem is *complete* iff it finds a solution to the Problem Statement 4.1 iff there exists one.

5 Adding Nonmasking Fault-Tolerance

Algorithm sketch. Since a nonmasking program is not required to satisfy its safety specification in the presence of faults, it only suffices to provide bounded-time recovery from the fault-span $S_p - S$ to the invariant S. We develop a general procedure that adds bounded-time recovery to a given region graph from any arbitrary given state predicate P to another state predicate Q within δ time units (i.e., $P \mapsto_{\leq\delta} Q$). Notice that bounded-time recovery from fault-span to the invariant can be formally defined by $\mathcal{R} \equiv (S_p - S) \mapsto_{\leq\delta} S$. The algorithm has four main steps. First, we transform the region graph to a weighted directed graph (called MaxDelay digraph [24]), in which the length of a path from vertex v_s to v_t is equivalent to the maximum delay for reaching the region that corresponds to v_t from the region that corresponds to v_s. We use this property to remove the computations that violate $P \mapsto_{\leq\delta} Q$. To this end, in Step 2, we rank vertices of the MaxDelay digraph by simply applying an *adjusted* Dijkstra's shortest path algorithms. For instance, suppose that a computation starts from a state $\sigma_0 \in P$. If a fault perturbs the program to a state σ_j where "something" should be *redone*, the maximum delay of that computation to reach Q is obviously increased. Hence, we adjust the length of the shortest path from σ_0 to Q such that the amount of time wasted by every occurrence of faults is considered (cf. Figure 1). In Step 3, we include regions and edges whose rank is at most the required response time δ. Then, in Step 4, we transform the synthesized MaxDelay digraph back into a region graph.
Construction of MaxDelay digraph. We now describe the procedure ConstructMaxDelayGraph that transforms a region graph to a MaxDelay digraph. The procedure takes a region graph $R(\mathcal{P})\langle S^r_p, \psi^r_p\rangle$ and a set f^r of fault edges as input, and constructs a MaxDelay digraph $G\langle V, A\rangle$ as follows. Vertices of G consist of the regions in $R(\mathcal{P})$.
Notation: We denote the weight of an arc (v_0, v_1) by $Weight(v_0, v_1)$. Let γ denote a bijection that maps each region $r \in S^r_p$ to its corresponding vertex in G; i.e., $\gamma(r)$ is a vertex of G that represents region r of $R(\mathcal{P})$. Also, let γ^{-1} denote the inverse of γ; i.e.,

$\gamma^{-1}(v)$ is the region of $R(\mathcal{P})$ that corresponds to vertex v in V. Let Γ be a function that maps a region predicate in $R(\mathcal{P})$ to the corresponding set of vertices of G and let Γ^{-1} be its inverse. Finally, for a boundary region r with respect to clock variable x, we denote the value of x by $r.x$ (equal to some nonnegative integer in $\mathbb{Z}_{\geq 0}$).
Arcs of G consist of the following:

- Arcs of weight 0 from v_0 to v_1, if $\gamma^{-1}(v_0) \rightarrow \gamma^{-1}(v_1)$ represents a jump transition in $R(\mathcal{P})$.
- Arcs of weight $c' - c$, where $c, c' \in \mathbb{Z}_{\geq 0}$ and $c' > c$, from v_0 to v_1, if $\gamma^{-1}(v_0)$ and $\gamma^{-1}(v_1)$ are both boundary regions with respect to clock variable x_i, such that $\gamma^{-1}(v_0).x_i = c$, $\gamma^{-1}(v_1).x_i = c'$, and there is a path in $R(\mathcal{P})$ from $\gamma^{-1}(v_0)$ to $\gamma^{-1}(v_1)$, which does not reset x_i.
- Arcs of weight $c' - c - \epsilon$, where $c, c' \in \mathbb{Z}_{\geq 0}$, $c' > c$, and $0 < \epsilon \ll 1$, from v_0 to v_1 , if (1) $\gamma^{-1}(v_0)$ is a boundary region with respect to clock x_i, (2) $\gamma^{-1}(v_1)$ is an open region whose time-successor $\gamma^{-1}(v_2)$ is a boundary region with respect to clock x_i, (3) $\gamma^{-1}(v_0) \rightarrow \gamma^{-1}(v_1)$ represents a delay transition in $R(\mathcal{P})$, and (4) $\gamma^{-1}(v_0).x_i = c$ and $\gamma^{-1}(v_2).x_i = c'$.
- Self-loop arcs of weight ∞ at vertex v, if $\gamma^{-1}(v)$ is an end region.

In order to compute the maximum delay between regions in P^r and Q^r, it suffices to find the longest distance between $\Gamma(P^r)$ and $\Gamma(Q^r)$ in G.

We now describe the procedure Add_BoundedRecovery (cf. Figure 2) in detail. Given a region graph $R(\mathcal{P})$, we first transforms it into a MaxDelay digraph $G\langle V, A\rangle$ (Line A1). Recall that, by Assumption 3.2, faults are detectable and $\mathcal{P}$ has a variable that shows how many faults have occurred in a computation. Thus, let $G^i\langle V^i, A^i\rangle$ be the portion of G, in which $n - i$ faults have occurred, where $0 \leq i \leq n$. More specifically, initially, a computation starts from the portion G^n where no faults have occurred. If a fault occurs in a computation that is currently in portion G^i, the computation will proceed in portion G^{i-1}. We use these portions to see whether it is possible to reach a vertex in $\Gamma(Q^r)$ from each vertex in $\Gamma(P^r)$ within δ time units.

Next, we rank vertices of all portions of G using a modified Dijkstra's shortest path algorithm, which takes state perturbations into account (lines A2-A9 and A22-A23). More specifically, since no faults occur in G^0, we first let the rank of each vertex $v \in V^0$ be the length of Dijkstra's shortest path from v to $\Gamma(Q^r)^0$ (Line A2). Now, let v_0 be a vertex in V^i where $1 \leq i \leq n$, and let v_1 be a vertex in V^{i-1}, such that

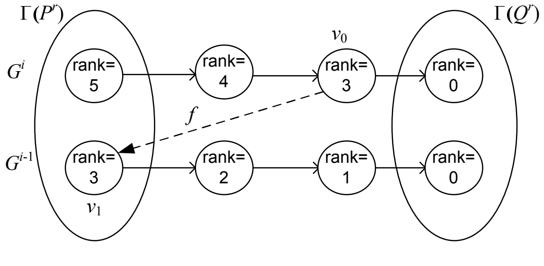

Fig. 1. Adjusted shortest path.

procedure Add_BoundedRecovery($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$: region graph, f^r: set of edges,
P^r, Q^r: region predicate, n, δ: **integer**)
// Adds bounded-time recovery from P^r to Q^r in the presence of f^r
{
step 1: $G\langle V, A\rangle :=$ ConstructMaxDelayGraph$(R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r)$; (A1)
Let $G^i\langle V^i, A^i\rangle$ be the portion of G, in which $(n-i)$ faults have occurred, where $0 \leq i \leq n$;
step 2: **for** each vertex $v \in V^0$: $Rank(v) :=$ Length of the shortest path from v to $\Gamma(Q^r)^0$; (A2)
for $i = 1$ **to** n (A3)
for each vertex $v_0 \in V^i$: (A4)
$V_f := \{v_1 \mid (v_1 \in V^{i-1} \;\wedge\; (\gamma^{-1}(v_0), \gamma^{-1}(v_1)) \in f^r)\}$; (A5)
if $V_f \neq \{\}$ **then** $MinRank(v_0) :=$ (A6)
$\max\{(Rank(v_1) + Weight(v_0, v_1))$ for all $v_1 \in V_f\}$; (A7)
else $MinRank(v_0) := 0$; (A8)
AdjustShortestPaths$(G^i\langle V^i, A^i\rangle, \Gamma(P^r)^i, \Gamma(Q^r)^i)$; (A9)
// Constructing a subgraph of each portion such that the longest distance between $\Gamma(P^r)$ and $\Gamma(Q^r)$ is at most δ and then adding the arcs and vertices that do not appear on paths from $\Gamma(P^r)$ to $\Gamma(Q^r)$
step 3: **for** $i = 0$ **to** n (A10)
$G'^i\langle V'^i, A'^i\rangle = \{\}$; (A11)
for each vertex $v \in \Gamma(P^r)^i$: (A12)
if $Rank(v) \leq \delta$ **then** (A13)
$\Pi :=$ the shortest path from v to $\Gamma(Q^r)^i$; (A14)
$V'^i := V'^i \cup \{u \mid u$ is on $\Pi\}$; (A15)
$A'^i := A'^i \cup \{a \mid a$ is on $\Pi\}$; (A16)
$A'^i := A'^i \cup \{(u,v) \mid (u,v) \in A^i \;\wedge\; (u \notin V'^i \;\vee\; (u \in \Gamma(Q^r)^i))\}$; (A17)
$V'^i := (V'^i \cup \{u \mid (\exists v : (u,v) \in A'^i \;\vee\; (v,u) \in A'^i)\})$; (A18)
// Transforming weighted digraph G into a region graph
step 4: $\psi_p'^r := \{(r_0, r_1) \mid (r_0, r_1) \in \psi_p^r \;\wedge\; (\gamma(r_0), \gamma(r_1)) \in A'\} \cup$
$\{(r_1, r_2) \mid (r_1, r_2) \in \psi_p^r \;\wedge\; (\gamma(r_1), \gamma(r_2)) \notin A' \;\wedge$
$\exists r_0 : Weight(\gamma(r_0), \gamma(r_1)) = 1 - \epsilon\}$; (A19)
$ns := \{r \mid \gamma(r) \in (V - V')\}$; (A20)
return $\psi_p'^r, ns$ (A21)
}
procedure AdjustShortestPaths$(G^i\langle V^i, A^i\rangle$: directed weighted graph, V_q: set of vertices)
// Adjusts the rank of each vertex based on the ranks computed in Add_BoundedRecovery
{
for each vertex $v \in V^i$ apply Dijkstra's shortest path with the following change:
if Dijkstra's shortest path computes a length less than $MinRank(v)$ **then**
$Rank(v) := MinRank(v)$; (A22)
else $Rank(v) :=$ length of Dijkstra's shortest path from v to V_q
using the assigned rank of other vertices (A23)
}
algorithm Add_Nonmasking($\mathcal{P}\langle S_p, \psi_p\rangle$:real-time program f :transitions, S: state predicate, n, δ: **integer**)
{
$R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, S^r, f^r :=$ ConstructRegionGraph$(\mathcal{P}\langle S_p, \psi_p\rangle, S, f)$; (B1)
$\psi_p^r := \psi_p^r \cup \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \notin S^r \wedge$
$\exists \rho_2 \mid \rho_2$ is a time-successor of ρ_0 : $(\exists \lambda \subseteq X : \rho_1 = \rho_2[\lambda - \{t\} := 0])\}$; (B2)
$\psi_p^r, ns :=$ Add_BoundedRecovery$(R(\mathcal{P})\langle S_p^r, \psi_{p_1}^r\rangle, f^r, S_p^r - S^r, S^r, n, \delta)$; (B3)
$rs := \{r_0 \mid \exists r_1, r_2, ...r_n : (\forall j : 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \wedge r_n \in (ns \cap P^r)\}$; (B4)
$rt := \{(r_0, r_1) \mid (r_0, r_1) \in \psi_p^r \;\wedge\; r_1 \in rs)\}$; (B5)
$S'^r, \psi_p'^r := S^r - rs, \psi_p^r - rt$; (B6)
$\psi_p'^r :=$ EnsureClosure(ψ_p^r, S'^r); (B7)
$\mathcal{P}'\langle S_p, \psi_p'\rangle, S' :=$ ConstructRealTimeProgram$(R(\mathcal{P})\langle S_p^r, \psi_p'^r\rangle, S'^r)$ (B8)
}
procedure EnsureClosure(ψ_p^r : set of edges, S^r : region predicate)
{ **return** $\psi_p^r - \{(r_0, r_1) \mid r_0 \in S^r \;\wedge\; r_1 \notin S^r\}\}$

Fig. 2. Adding Bounded-Time Recovery/Nonmasking Fault-Tolerance.

$(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ is a fault edge in $R(\mathcal{P})$ and both v_0 and v_1 are on a path from $\Gamma(P^r)$ to $\Gamma(Q^r)$. There exist two cases: (1) the fault edge $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ decreases or does not change the computation delay, i.e, the shortest distance from v_1 to $\Gamma(Q^r)^{i-1}$ is less than or equal to the shortest distance from v_0 to $\Gamma(Q^r)^i$, and (2) the fault edge $(\gamma^{-1}(v_0), \gamma^{-1}(v_1))$ increases the computation delay, i.e., the shortest distance from v_1 to $\Gamma(Q^r)^{i-1}$ is greater than the shortest distance from v_0 to $\Gamma(Q^r)^i$ (cf. Figure 1 for an example). While the former case does not cause violation of $P \mapsto_{\leq\delta} Q$ in the presence of faults, the later may do. Hence, the rank of $v_0 \in V^i$ must be at least the rank of $v_1 \in V^{i-1}$. Moreover, if there exist multiple fault edges at $\gamma^{-1}(v_0)$ then we take the maximum rank (Line A7). After computing the rank of vertices from where faults may occur, we adjust the rank of the rest of vertices from where faults do not occur by invoking the procedure AdjustShortestPath (Line A9).

Now, for each portion G^i, we construct a subgraph of G^i whose longest distance from each vertex in $\Gamma(P^r)^i$ to $\Gamma(Q^r)^i$ is at most δ as follows (lines A11-A16). We begin with an empty digraph $G'^i\langle V'^i, A'^i\rangle$ and we first include the shortest paths from each vertex $v \in \Gamma(P^r)^i$ to $\Gamma(Q^r)^i$, provided $Rank(v) \leq \delta$ (lines A13-A16). Next, we include the remaining arcs and vertices in G'^i, so that no arcs of the form (v_0, v_1), where v_0 is on a path from $\Gamma(P^r)^i$ to $\Gamma(Q^r)^i$ are added (lines A17-A18).

Now, we transform the digraph G' back into a region graph (Line A19). Finally, we return the set ψ'^r_p of edges from where $P \mapsto_{\leq\delta} Q$ is not violated even in the presence of faults, and the set ns of regions from where $P \mapsto_{\leq\delta} Q$ may be violated in the presence of faults (lines A20-A21).

Using Add_BoundedRecovery to Add Nonmasking Fault-Tolerance. In order to add nonmasking fault-tolerance with bounded-time recovery δ, we first transform the real-time program $\mathcal{P}\langle S_p, \psi_p\rangle$, invariant S, and the set of fault transitions f into a region graph $R(\mathcal{P})\langle S^r_p, \psi^r_p\rangle$, region invariant S^r, and fault edges f^r by invoking the procedure ConstructRegionGraph (Line B1), as described in Subsection 2.3. Next, in order to ensure that S' is reachable from all the states in $S_p - S'$, we add *recovery edges* that start from each region in $S^r_p - S^r$ and go to regions where the time monotonicity condition is preserved, i.e., time is not decreased (Line B2). Notice that the algorithm allows arbitrary clock resets (except the clock that keeps track of the recovery time time δ) during recovery, which is fine according to the definition of nonmasking fault-tolerance (such "new" clock resets occur only in states outside the invariant). Then we invoke the procedure Add_BoundedRecovery. This invocation identifies the set rs of regions and the set rt of edges from where faults alone may violate $\mathcal{R}$ (lines B4-B5). Then, it removes such regions (respectively, edges) from S^r (respectively, ψ^r_p). Finally, the algorithm ensures the closure of the invariant (Line B7) and transforms the synthesized region graph $R(\mathcal{P}')$ back to a real-time program $\mathcal{P}'$ (Line B8).

6 Adding Soft-Failsafe Fault-Tolerance

As mentioned in Subsection 2.2, the safety specification identifies a set Σ_{bt} of bad transitions and a conjunction Σ_{br} of multiple bounded response properties. Also, recall that in the presence of faults, a soft-failsafe program is required to maintain Σ_{bt} only.

Algorithm sketch. We adapt the proposed algorithm in [6], which adds failsafe fault-tolerance to *untimed* programs. Intuitively, our algorithm, consists of three main steps. First, we prohibit the program from reaching the set ms of states from where a sequence of faults takes the program to a state where safety (Σ_{bt}) is violated. Since our goal is to synthesize a *maximal* program, we find ms by computing the smallest fixpoint of states from where safety may be violated. In step 2, after removing ms from the program invariant S, we make sure that this removal do not create new finite computations in the absence of faults. To this end, we remove deadlock states from the invariant which is in turn computing the largest fixpoint of the invariant. Finally, in step 3, we ensure that removal of transitions from where safety may be violated does not violate the closure of the output program.

We now describe the algorithm Add_SoftFalisafe (cf. Figure 3). We first transform the program $\mathcal{P}$ into its region graph $R(\mathcal{P})$ (Line C1). Then, the algorithm adds failsafe fault-tolerance to $R(\mathcal{P})$, so that no edge of Σ^r_{bt} occurs in computations of $R(\mathcal{P})$ in the presence of faults by invoking the procedure Add_UntimedFailsafe (Line C2). This procedure first finds the set ms of regions and the set mt of edges from where safety of $\mathcal{P}$ may be violated by faults alone (lines E1-E2). Next, it removes such regions (respectively, edges) from the region invariant S^r (respectively, set of edges ψ^r_p) of $R(\mathcal{P})$. Then, it removes deadlock regions from S^r (Line E3), ensures the closure of ψ^r_p in S^r (Line E5), and returns a failsafe region graph $R(\mathcal{P}')\langle S^r_p, \psi'^r_p\rangle$ (Line E6). Finally, we transform the region graph $R(\mathcal{P}')$ back into a real-time program $\mathcal{P}'$ (Line C3) as described in Subsection 2.3.

7 Adding Hard-Failsafe Fault-Tolerance with one Bounded Response Property

In this section, we present our algorithm Add_HardFailsafe (cf. Figure 3) for the case where the synthesized hard-failsafe program is required to satisfy at most one bounded response property in the presence of faults, i.e., $\Sigma_{br} \equiv P \mapsto_{\leq\delta} Q$.

Algorithm sketch. Intuitively, the algorithms works in five main steps. First, we add soft-failsafe to $R(\mathcal{P})$ to ensure that a transition in Σ_{bt} occurs in no computation of $\mathcal{P}'$. Note that, the outcome of adding soft-failsafe is a maximal program and every transition that is removed by Add_SoftFailsafe has to be removed, i.e., such transitions cannot be in any fault-tolerant program that satisfies the constraints of Problem Statement 4.1. In Step 2, we remove the behaviors that violate the bounded response property $\Sigma_{br} \equiv P \mapsto_{\leq\delta} Q$ in the presence of faults using the procedure Add_BoundedRecovery. In step 3, we remove deadlock states due to removal of states and transitions in step 2. In Step 4, if a state $\sigma_1 \in Q$ is removed and some state, say σ_0, in P uses σ_1 to satisfy $P \mapsto_{\leq\delta} Q$ then another path from σ_0 must be provided to satisfy $P \mapsto_{\leq\delta} Q$. Hence, we remove σ_1 from Q and repeat steps 2, 3, and 4 until no such Q-states exist. Finally, in Step 5, we ensure the closure of the output program.

We now describe the pseudo-code of the algorithm. In order to ensure that $\mathcal{P}'$ maintains Σ_{bt}, we first add soft-failsafe fault-tolerance to $R(\mathcal{P})$ (Line D1). Next, we transform $\mathcal{P}$ into its region graph $R(\mathcal{P})$ (Line D2). Next, we modify $R(\mathcal{P})$, such that any computation that starts from a region in P^r, reaches a region in Q^r in at most δ time

algorithm Add_SoftFailsafe($\mathcal{P}\langle S_p, \psi_p\rangle$:real-time program f :transitions, S: state predicate, Σ_{bt}: specification)
{
$R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, S^r, f^r, \Sigma_{bt}^r :=$ ConstructRegionGraph($\mathcal{P}\langle S_p, \psi_p\rangle, S, f, \Sigma_{bt}$); (C1)
$\psi_p'^r, S'^r :=$ Add_UntimedFailsafe$(R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r, S^r, \Sigma_{bt}^r)$; (C2)
$\mathcal{P}'\langle S_p, \psi_p'\rangle, S' :=$ ConstructRealTimeProgram$(R(\mathcal{P})\langle S_p^r, \psi_p'^r\rangle, S'^r)$; (C3)
return $\mathcal{P}'\langle S_p, \psi_p'\rangle, S'$; (C4)
}
algorithm Add_HardFailsafe($\mathcal{P}\langle S_p, \psi_p\rangle$:real-time program f :transitions,
S, P, Q: state predicate, Σ_{bt}: specification, n, δ: **integer**)
{
step 1: $\mathcal{P}\langle S_p, \psi_p\rangle, S :=$ Add_SoftFailsafe$(R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r, S^r, \Sigma_{bt}^r)$; (D1)
$R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, S^r, P^r, Q^r, f^r, \Sigma_{bt}^r :=$
ConstructRegionGraph$(\mathcal{P}\langle S_p, \psi_p\rangle, S, P, Q, f, \Sigma_{bt})$; (D2)
repeat
IsQRemoved := *false*; (D3)
step 2: $\psi_p^r := \psi_p^r \cup \{((s_0, \rho_0), (s_1, \rho_1)) \mid (s_0, \rho_0) \notin S^r \wedge$
$\exists\rho_2 \mid \rho_2$ is a time-successor of ρ_0 : $(\exists\lambda \subseteq X : \rho_1 = \rho_2[\lambda - \{t\} := 0])\} - mt$; (D4)
$\psi_p^r, ns :=$ Add_BoundedRecovery$(R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle, f^r, P^r, Q^r, n, \delta)$; (D5)
$rs := \{r_0 \mid \exists r_1, r_2, ...r_n :$
$(\forall j : 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \wedge r_n \in (ns \cap P^r)\}$; (D6)
$rt := \{(r_0, r_1) \mid (r_0, r_1) \in \psi_{p_1}^r \wedge r_1 \in rs)\}$; (D7)
step 3: $S'^r :=$ RemoveDeadlocks$(S^r - (ns \cup rs), \psi_p^r - rt)$; (D8)
if $(S'^r = \{\})$ **then**
`declare no hard-failsafe` f`-tolerant program` $\mathcal{P}'$ `exists`; **exit**; (D9)
step 4: **if** $(Q^r \cap (S^r - S'^r) \neq \{\})$ **then** (D10)
IsQRemoved := *true*; (D11)
$S^r := S'^r$; (D12)
$\psi_p^r := \psi_p^r - \{(r, r_0), (r_0, r) \mid r_0 \in Q^r \cap (S^r - S'^r)\}$; (D13)
$Q^r := Q^r \cap (S^r - S'^r)$; (D14)
until (*IsQRemoved* = *false*);
step 5: $\psi_p'^r :=$ EnsureClosure(ψ_p^r, S'^r); (D15)
$\mathcal{P}'\langle S_p, \psi_p'\rangle, S' :=$ ConstructRealTimeProgram$(R(\mathcal{P})\langle S_p^r, \psi_p'^r\rangle, S'^r)$ (D16)
}
procedure Add_UntimedFailsafe($R(\mathcal{P})\langle S_p^r, \psi_p^r\rangle$: region graph, f^r : set of edges,
S^r : region predicate, Σ_{bt}^r : specification)
{
step1: $ms := \{r_0 \mid \exists r_1, r_2, ...r_n :$
$(\forall j \mid 0 \leq j < n : (r_j, r_{j+1}) \in f^r) \wedge (r_{n-1}, r_n) \in \Sigma_{bt}^r\}$; (E1)
$mt := \{(r_0, r_1) \mid (r_1 \in ms) \vee ((r_0, r_1) \in \Sigma_{bt}^r)\}$; (E2)
step2: $S^r :=$ RemoveDeadlocks$(S^r - ms, \psi_p^r - mt)$; (E3)
if $(S^r = \{\})$ **then**
`declare no soft/hard-failsafe` f`-tolerant program` $\mathcal{P}'$ `exists`; **exit**; (E4)
step3: $\psi_p^r :=$ EnsureClosure$(\psi_p^r - mt, S^r)$; (E5)
return ψ_p^r, S^r (E6)
}
procedure RemoveDeadlocks(S^r : region predicate, ψ_p^r : set of edges)
// Returns the largest subset of S^r from where all computations of $R(\mathcal{P})$ are infinite
{
while $(\exists r_0 \mid r_0 \in S^r : (\forall r_1 \in S^r : (r_0, r_1) \notin \psi_p^r))$
$S^r := S^r - \{r_0\}$;
return S^r
}

Fig. 3. Adding Failsafe Fault-Tolerance.

units even in the presence of faults. Towards this end, we compute the set of regions and edges from where Σ_{br} is maintained (lines D3-D14). Precisely, in order to ensure that Q is reachable from all the states in $P \wedge \neg S$, we first include edges that start from each region in $S_p^r - S^r$ and go to regions where the time monotonicity condition is preserved, i.e., time is not decreased (Line D4). Notice that the algorithm allows arbitrary clock resets as long as safety is not violated (by excluding the edges in mt). Then, we invoke the procedure Add_BoundedRecovery to ensure that $P \mapsto_{\leq \delta} Q$ is maintained in the presence of faults (Line D5). Then, we identify the set rs of regions and rt of transitions from where Σ_{br} may be violated (lines D6-D7). We remove such regions and edges along with the deadlock regions from S^r in the same fashion that we did for adding soft-failsafe (Line D8). However, we need to consider a special case where a region, say r_1, in Q^r becomes a deadlock region. In this case, it is possible that all the regions along the paths that start from a region, say r_0, in P^r and end in r_1 become deadlock regions. Hence, we need to find another path from r_0 to a region in Q^r other than r_1. Thus, in this case, we remove r_1 (and similar regions) from S^r and Q^r and start over (lines D10-D14). Finally, the algorithm ensures closure of the invariant (Line D15) and transforms the synthesized region graph $R(\mathcal{P}')$ back to a real-time program $\mathcal{P}'$ (Line D16).

Theorem 7.1. The algorithms Add_Nonmasking and Add_Soft/HardFalisafe are sound and complete. □

Theorem 7.2. The problem of adding nonmasking and soft/hard-failsafe fault-tolerance to a real-time program, where the synthesized program is required to satisfy at most one bounded response property in the presence of faults, is PSPACE-complete in the size of the input program. □

8 Conclusion

In this paper, we focused on the problem of automatic addition of fault-tolerance to real-time programs. We considered three levels of fault-tolerance, namely failsafe, nonmasking, and masking. For failsafe and masking, we proposed two cases, soft and hard, based on satisfaction of timing constraints in the presence of faults. We first introduced a generic framework to formally define the notions of faults and fault-tolerance in the context of real-time programs. Then, we presented sound and complete algorithms for transforming fault-intolerant real-time programs into soft-failsafe and nonmasking fault-tolerant programs. We also proposed a sound and complete algorithm that synthesizes hard-failsafe fault-tolerant real-time programs, where the fault-tolerant program is required to satisfy at most one bounded response property in the presence of faults. The complexity of our algorithms are in polynomial time in the size region graphs. The results on synthesis of soft/hard-masking fault-tolerance are presented in a technical report [19].

References

1. M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Transations on Computers*, 47(10):1102–1112, 1998.

2. D. Mossé, R. G. Melhem, and S. Ghosh. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Transactions on Software Engineering*, 29(8):752–767, 2003.
3. B. Bonakdarpour and S. S. Kulkarni. Automated incremental synthesis of timed automata. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 2006.
4. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
5. A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, 2005.
6. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.
7. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *20th Symposium on Reliable Distributed Systems (SRDS)*, pages 130–140, 2001.
8. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 209–219, 2004.
9. P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, 2004.
10. 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.
11. 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.
12. D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *19th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 571–582, 2002.
13. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.
14. L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *14th International Conference on Concurrency Theory (CONCUR)*, 2003.
15. M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.
16. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
17. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
18. Y. Abdeddam. *Scheduling with Timed Automata*. PhD thesis, INPG, Grenoble, November 2002.
19. B. Bonakdarpour and S. S. Kulkarni. Automatic addition of fault-tolerance to real-time programs. Technical Report MSU-CSE-06-13, Department of Computer Science and Engineering, Michigan State University, 2006.
20. 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.
21. T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.
22. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
23. 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.
24. C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *Computer-Aided Verificaion (CAV)*, pages 399–409, 1991.