Zone-based Synthesis of Timed Models with Strict Phased Fault Recovery

Fathiyeh Faghih and Borzoo Bonakdarpour

School of Computer Science, University of Waterloo, Canada

TECHNICAL REPORT CS-2013-05

Abstract. In this paper, we focus on efficient synthesis of fault-tolerant timed models from their fault-intolerant version. Although the complexity of the synthesis problem is known to be polynomial time in the size of the time-abstract bisimulation of the input model, the state of the art currently lacks synthesis algorithms that can be efficiently implemented. We propose an algorithm that takes a timed automaton, a set of fault actions, and a set of safety and bounded-time response properties as input, and utilizes a space-efficient symbolic representation of the timed automaton (called the *zone graph*) to synthesize a fault-tolerant timed automaton as output. The output automaton satisfies strict phased recovery, where it is guaranteed that the output model behaves similarly to the input model in the absence of faults and in the presence of faults, fault recovery is achieved in two phases, each satisfying certain safety and timing constraints. Our algorithm is fully implemented and we report encouraging experimental results.

1 Introduction

[1]Dependability and time-predictability are two vital properties of most embedded (especially, safety/mission-critical) systems.time-sensitive computing systems. Consequently, providing *fault-tolerance* and meeting *timing constraints* are two inevitable aspects of dependable real-time embedded systems. However, these two features have conflicting natures; i.e., fault-tolerance deals with unanticipated faults, while meeting time constraints requires time predictability. This conflict inevitably makes design and analysis of fault-tolerant real-time systems a tedious and error-prone task. Hence, it is highly desirable to have access to techniques that automatically generate correct-by-construction models that ensure fault-tolerance and meet timing constraints simultaneously.

Automated *synthesis* is a rigorous but highly complex method to generate models that are correct by construction. There are different approaches for model synthesis depending upon the input. Examples include synthesis from a temporal logic specification [3, 16] along with quantitative objectives [6, 14], program sketching [24], and controller synthesis [22, 23]. Automated addition of fault-tolerance is also a technique for synthesizing a fault-tolerant model from its fault-intolerant version [8,9,18]. This line of work is in spirit close to controller synthesis, where faults can be modeled as uncontrollable transitions[2]. However, there are subtle differences (e.g., synthesis of recovery paths), which make the

[1] The full version of this paper can be found in our technical report [17]. Due to space limitation, a running example and an additional case study are omitted from this version of the paper.

[2] We emphasize that representation of faults as transitions is possible for different types of faults (e.g., stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss,etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults [7].

problem more complex than conventional controller synthesis. Another important difference is in the fact that in many commonly considered systems, fault recovery has to be achieved in multiple (possibly ordered) *phases*, each satisfying certain constraints. For example, in a traffic signal controller, if the controller detects a fault, all signals should first go red immediately to prevent catastrophic consequences (i.e., phase 1) before final recovery to its normal behavior (i.e., phase 2).

In the context of synthesizing timed models that provide bounded-time phased fault recovery, let Q and P be two predicates that should be reached in phase 1 and 2 of recovery within different time bounds, respectively. In [7], the authors have shown that if Q is not required to be closed in the execution of recovery transitions, then synthesizing a timed automaton [2] with 2-phase recovery is NP-complete in the size of the detailed region graph [2] of the input automaton[3]. On the contrary, if the closure of Q is required and, moreover, $P \subseteq Q$, then the synthesis problem can be solved in polynomial time. The polynomial-time algorithm presented in [7] to solve the latter problem is only an evidence for proving the complexity of the problem and is not an efficient practical solution with potential for implementation. This is simply because the size of a detailed region graph grows incredibly huge even for small models.

With this motivation, in this paper, we propose a time- and space-efficient algorithm for synthesizing timed automata that provide 2-phase recovery, where Q is required to be closed and $P \subseteq Q$, while no new behaviors are added in the absence of faults. The latter is guaranteed by only augmenting the semantic model with safe recovery paths in the presence of faults and adding no transitions that originate from states that can be reached only in the absence of faults. For space efficiency, we utilize the notion of *zone graphs* [15] developed as finite representation of timed automata. Although there is work on synthesis of timed models using zone graphs (most notably the tool UPPAAL-Tiga [4]), the state of the art currently lacks two ingredients to make synthesis of fault-tolerant timed models possible: (1) zone-based controller synthesis for bounded response properties of the form $Q \mapsto_{\leq \delta} P$; i.e., when Q becomes true, P should become true within δ time units, and (2) zone-based addition of safe recovery paths that do not exist in the original intolerant model. This is a challenging problem, as adding zones without considering their properties might lead to generating deadlock computations.

Our fully implemented algorithm addresses both aforementioned problems. Given a timed automaton, a set of fault actions, and constraints of 2-phase recovery, our implementation first generates a zone graph using the tool IF [11]. Then, it adds paths for each phase of recovery by incorporating its constraints. Finally, it ensures deadlock freedom by implementing a global fixpoint computation and repair. Our experiments show that the performance of the proposed synthesis algorithm can compete with model checking, where the synthesis time is proportional to the corresponding verification time (i.e., zone graph generation time for the input model).

Organization. The rest of the paper is organized as follows. In section 2, we present the preliminary concepts. Section 3 describes timed automata with faults and the notion of strict 2-phase recovery. Section 4 formally states the synthesis problem, while Section 5 presents our zone-based synthesis algorithm. We describe our implementation and experimental results in Section 6.Related work is discussed in Section 7. Finally, we make concluding remarks in Section 8. For reasons of space, all proofs appear in the appendices.

[3] A detailed region graph is a finite bisimilar representation of a timed automaton.

2 Preliminaries

In this section, we present the preliminary concepts on timed automata and specifications in Subsections 2.1 and 2.2, respectively.

2.1 Timed Automata with Deadlines (TAD) [2, 10]

Syntax Let $X = \{x_1, x_2, ..., x_m\}$ be a finite set of *clock variables* that range over real numbers $\mathbb{R}_{\geq 0} \cup \{-1\}$. The value -1 identifies a *disabled* clock variable. The set Φ of all *clock constraints* over X is inductively defined as follows:

$$p ::= x \sim n \mid p \wedge p \mid \neg p$$

where n is a constant non-negative integer, and $\sim \in \{<, \leq, >, \geq\}$. Let V be a set of finite-domain *discrete variables*. We denote the set of all *guards* (i.e., Boolean expressions) over V by G_D.

Definition 1. *A* timed automaton with deadline *is a tuple* $TAD = (L, l_0, V, U, X, E)$*, where*

- L *is a finite set of* locations
- $l_0 \in L$ *is the* initial location
- V *is a finite set of* discrete variables
- U *is a finite set of* update functions
- X *is a finite set of clock variables and*
- $E \subseteq L \times U \times G_D \times \Phi \times \Phi \times 2^X \times 2^X \times L$ *is a finite set of timed* switches.

Each timed switch is of the form $(l, u, g_d, g_c, d, (X_{res}, X_{dis}), l')$*, where* X_{res} *is a set of clocks to be reset,* X_{dis} *is a set of clock variables being disabled, such that* $X_{res} \cap X_{dis} = \{\}$*,* $g_c \in \Phi$ *is a clock constraint, and* $d \in \Phi$ *is the transition delay, such that* $d \Rightarrow g_c$*.* □

In Definition 1, delay d determines the urgency of a switch. There are three different types of delays [10].Intuitively, when $d = g_c$, the switch is called *eager*. An enabled eager switch cannot be delayed and, hence, does not let time progress before its execution. If $d = \mathit{false}$, then the switch is *lazy*, meaning that whenever it gets enabled, its execution can be delayed by letting time progress. This delay may even result in disabling the transition. In a *delayable* switch, d is the falling edge of a right-closed guard g_c; *i.e.,* whenever a delayable switch is enabled, its execution can be delayed as long as the associated guard remains true.

Semantics In the following, we use val_d to denote a function that maps each $v \in V$ to a value in its finite domain Dom_v, and is called a *valuation* of discrete variables. Likewise, val_c denotes a *clock valuation*, which is a function that maps each clock variable $x \in X$ to a value in $\mathbb{R}_{\geq 0} \cup \{-1\}$. An *update function* $u \in U$, is a function $Dom_{v_1} \times \ldots \times Dom_{v_{|V|}} \rightarrow Dom_{v_1} \times \ldots \times Dom_{v_{|V|}}$ that maps each valuation val_d to a valuation val'_d. We denote the fact that a (clock or discrete) valuation val satisfies a guard g by $val \models g$. Each element of a tuple denoting a switch e is presented by the name of the element subscripted by e. For example, u_e denotes the update function of the switch e.

The *semantic model* of a TAD is a tuple $\mathcal{SM} = (S, s_0, T)$, where

- S is the *state space* of the semantic model. Each *state* is a tuple (l, val_d, val_c), where $l \in L$ is a location, and val_d and val_c are discrete and clock valuations, respectively.

- $s_0 = (l_0, (val_d)_0, \overrightarrow{0})$ is the *initial state*, where l_0 is the initial location, $(val_d)_0$ is a valuation in which all discrete variables are initialized to some value in their domains, and $\overrightarrow{0}$ denotes the clock valuation with all clocks being set to zero.
- T is the set of *transitions* on S. In order to define T, we first identify the clock valuations from where time can progress from a location l and valuation val_d. Let E_l be the set of switches originating from l. We define $c(l, val_d)$ as the set of clock valuations:

$$c(l, val_d) = \{val_c \mid \neg \bigvee_{e \in E_l} ((val_c \models d_e) \ \wedge \ (val_d \models (g_d)_e))\}$$

and is called the *time progress condition* of location l and valuation val_d. For $\delta \in \mathbb{R}_{\geq 0}$, we write $val_c + \delta$ to denote $val_c(x) + \delta$ for every clock variable $x \in X$, if $x \neq -1$ (i.e., time does not advance for disabled clocks). The set T of transitions in the semantic model is classified as follows:

Immediate Transitions A transition $(l, val_d, val_c) \ \rightarrow \ (l', val'_d, val_c[X_{res}, X_{dis}])$ exists in T iff there exists a switch $(l, u, g_d, g_c, d, (X_{res}, X_{dis}), l') \in E$, such that $(val_c \models g_c) \ \wedge \ (val_d \models g_d)$, where $u(val_d) = val'_d$, and $val_c[X_{res}, X_{dis}]$ is the valuation val_c, where
- for each $x \in X_{res}$, we have $val_c(x) = 0$
- for each $x \in X_{dis}$, we have $val_c(x) = -1$
- the value of other clock variables are unchanged.

The set of immediate transitions in T is denoted by T_{imm}.

Delay transitions A transition $(l, val_d, val_c) \rightarrow (l, val_d, val_c + \delta)$ exists in T iff $\forall t < \delta : (val_c + t) \in c(l, val_d)$. The set of delay transitions in T is denoted by T_d.

Example We use the following running example to describe the concepts throughout the paper. Consider two processes that execute in mutual exclusion using a shared memory location. To coordinate their execution, one of the processes is the master process (illustrated in Figure 1). The automaton has three locations, execution (initial location), cleanup, and waiting, a clock variable x, and a discrete variable $token$ shared between the processes. The clock constraint of switches are placed in [] and a switch delay is identified by $\{\}$.

The master process stays in execution for 1 to 2 time units. Then, it resets x, toggles the value of $token$, and goes to cleanup, where it can spend another 1 to 2 time units for garbage collection. Changing the value of the shared variable allows the slave process (not shown here) to start execution. Then, the master process goes to location waiting, where it waits for the slave process execution to finish. When the value of x is between 3 to 4 time units, it again toggles the value of $token$, so that the slave process stops execution, and reaches location cleanup. In this location, the master process does the garbage collection for the slave, and also ensures that the slave process has noticed the change in $token$. The master process subsequently moves to location execution.

2.2 Specification

In this section, we present the notion of specification and what it means for a timed automaton to satisfy a specification. First, we define *state predicates* on a timed automaton.

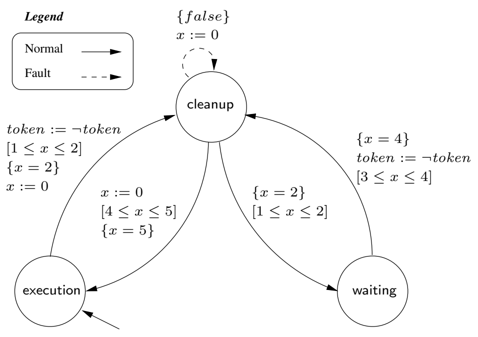

Fig. 1. An example of a timed automaton with deadline augmented with one fault switch

Definition 2. *A* state predicate *SP of a semantic model $\mathcal{SM} = (S, s_0, T)$ is a subset of S, where in the corresponding Boolean expression, each clock variable is only compared with non-negative integers.* □

In other words, a state predicate must be definable by the syntax of clock constraints as defined in Subsection 2.1.

Definition 3. *A* computation *of a semantic model $\mathcal{SM} = (S, s_0, T)$ is a finite or infinite sequence of states of the form: $\overline{s} = (s_0, \tau_0) \rightarrow (s_1, \tau_1) \rightarrow \ldots$ iff:*

- *for all $i \in \mathbb{Z}_{\geq 0}$: $(s_i, s_{i+1}) \in T$*
- *the sequence $\tau_0, \tau_1, \ldots$ (called the* global time*), satisfies the following conditions:*
  - monotonicity*: for all $i \in \mathbb{Z}_{\geq 0}, \tau_i \leq \tau_{i+1}$*
  - divergence*: if $\overline{s}$ is infinite, for all $t \in \mathbb{R}_{\geq 0}$, there exists $i \in \mathbb{Z}_{\geq 0}$, such that $\tau_i \geq t$*
  - consistency*: for all $i \in \mathbb{Z}_{\geq 0}$, if (s_i, s_{i+1}) is a delay transition in T, such that $s_i = (l, val_d, val_c)$, $s_{i+1} = (l, val_d, val_c + \delta)$, then $\tau_{i+1} - \tau_i = \delta$, and if (s_i, s_{i+1}) is an immediate transition in T, then $\tau_{i+1} = \tau_i$.* □

Observe that Definition 3 is tied with a semantic model. In general, this is not the case and a computation is a timed state sequence that can satisfy a subset of the three constraints in Definition 3.

We are now ready to define the notion of specifications and what it means for a timed automaton to satisfy a specification.

Definition 4. *A* specification *(or* property*) is a set of infinite computations that satisfy time-monotonicity and divergence [19].* □

Definition 5. *A state predicate SP is* closed *in a set of transitions T, iff*

- *if an immediate transition in T originates from SP, it terminates in SP*
- *if a delay transition in T with duration δ originates in state $s \in SP$, then for all $\delta' \leq \delta$, a delay transition with duration δ' that starts in s also terminates in a state in SP.* □

Definition 6. *Let TAD be a timed automaton with semantic model $\mathcal{SM} = (S, s_0, T)$ and SP be a state predicate of TAD. We write $TAD \models_{SP} SPEC$ (read TAD* satisfies *$SPEC$* from *SP), iff (1) SP is closed in T, and (2) every computation of TAD that starts from SP is in $SPEC$.* □

The reason for defining satisfaction 'from' a state predicate is due to the fact that when we add fault transitions to a model, the closure of its normal behavior is not ensured. This notion of normal behavior is captured by a state predicate called the set of *legitimate states* defined next.

Definition 7. *Let TAD be a timed automaton and LS be a nonempty state predicate of TAD. We say that LS is a set of* legitimate states *of TAD iff $TAD \models_{LS} SPEC$.* $\Box$

Definition 8. *Let P and Q be state predicates and $\delta \in \mathbb{R}_{\geq 0}$. A* bounded response *property is of the form $P \mapsto_{\leq \delta} Q$, and defines computations $\overline{s} = (s_0, \tau_0) \to (s_1, \tau_1) \to \ldots$, where for all $i \geq 0$, if $s_i \in P$, then there exists $j \geq i$, such that $s_j \in Q$ and $\tau_j - \tau_i \leq \delta$.* $\Box$

In this paper, our notion of specification consists oftwo parts: (1) a *safety specification*, and (2) a *liveness specification* [1, 19]. Roughly speaking, our notion of safety is characterized by a set of unsafe timing independent transitions and a set of bounded-time response properties.

Definition 9. *A* safety specification *consists of two parts:*

1. Timing-independent Safety*: Specified by a set of immediate* bad transitions *bt. The specification in which each computation has no bad transitions is denoted by $SPEC_{\overline{bt}}$.*
2. Timing Constraint*: Denoted by $SPEC_{\overline{br}}$ is the conjunction $\bigwedge_{i=1}^{m}(P_i \mapsto_{\leq \delta_i} Q_i)$.* $\Box$

A bad transition that can be specified by its target state only defines a set of *bad states*.

Definition 10. *A* liveness specification *$SPEC$ is a set of computations with this condition: for each finite computation $\overline{\alpha}$, there exists a nonempty suffix $\overline{\beta}$, such that $\overline{\alpha}\overline{\beta} \in SPEC$.* $\Box$

Following [1, 19], liveness specification is included in all specifications and, hence, it is not repeated in the specification representation.

Example. Consider the timed automaton in Figure 1. The timing independent safety specification for mutual exclusion between the two processes is characterized by:

$$bt = \{(s_0, s_1) \mid s_1 \models (\mathsf{execution} \wedge (token = 1))\}$$

which requires the master process not to be in location execution, when the value of $token$ is 1. The set of legitimate states of this example is specified using the following expression:

$$\begin{aligned} LS \equiv\ & ((\mathsf{execution}) \Rightarrow ((x \leq 2) \wedge (token = 0))) \wedge \\ & ((\mathsf{cleanup}) \quad \Rightarrow (((x \leq 2) \wedge (token = 1)) \vee \\ & \qquad\qquad\qquad ((3 \leq x \leq 5) \wedge (token = 0))) \wedge \\ & ((\mathsf{waiting}) \quad \Rightarrow ((1 \leq x \leq 4) \wedge (token = 1)) \end{aligned}$$

It is straightforward to see that starting from any state in LS, execution of *normal* switches of the timed automaton in Figure 1 results in a state in LS and a transition in $SPEC_{bt}$ will never execute.

3 Timed Automata with Faults and Strict 2-Phase Fault Recovery

In this section, we present the notions of faults and strict 2-phase fault recovery [7].

3.1 Fault Model

A *fault* is systematically represented as a transition. Fault representation with a transition is possible for different types of faults (e.g., stuck-at, crash, fail-stop, timing, performance, Byzantine, message loss, etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults [7].

Given a semantic model $\mathcal{SM} = (S, s_0, T)$, a set F of faults is a subset of all possible immediate transitions[4]. In other words, $F \subseteq (S \times S)_{imm}$, where

$$(S \times S)_{imm} = \{(l, val_d, val_c) \to (l', val'_d, val_c[X_{res}, X_{dis}]) \mid \\ (l, val_d, val_c), (l', val'_d, val_c[X_{res}, X_{dis}]) \in S \wedge X_{dis} = \emptyset\}$$

Similar to the notion of legitimate states for a timed automaton in the absence of faults, we introduce the notion of *fault-span* to reason about the behavior of a timed automaton in the presence of faults.

Definition 11. *For a semantic model $\mathcal{SM} = (S, s_0, T)$, legitimate states LS, and a set F of faults, a state predicate FS is a* fault-span *or F*-span *of the model $\mathcal{SM}$ from LS iff (1) LS $\subseteq$ FS, and (2) FS is closed in $T \cup F$.* $\square$

Hence, a fault-span is a state predicate up to which (but not beyond which) faults can perturb the state of a system. In order to distinguish the transitions/switches defined in the given timed automaton and faults, in the remainder of the paper, we call the former *normal* transitions/switches.

Example. In Figure 1, the fault switch introduced in location cleanup, resets clock variable x at any time. Notice that if x gets reset when $x \leq 2$, then this fault starts and ends within the legitimate states. However, if $3 \leq x \leq 5$ and x gets reset, then the fault leads the execution to a state outside the legitimate states. The delay of the fault switch is set to lazy, since it does not impose any constraints on time progress. Observe that, if a computation starts from a state in LS where $3 \leq x \leq 5$ and $token = 0$, when the fault occurs, after 1 to 2 time units, the computation goes to waiting and subsequently to cleanup where $token$ gets toggled (with value 1). The next transition of the computation is a bad transition, as the model goes to execution location, while $token = 1$. This clearly violates the safety specification.

3.2 Strict 2-phase Fault Recovery

Intuitively, in *strict 2-phase recovery* [7], when the state of a system is perturbed by faults, the system is required to either directly return to its legitimate states LS within $\theta \in \mathbb{Z}_{\geq 0}$ time units, or, if direct recovery is not feasible, then it should first reach an *intermediate* recovery predicate Q within $\theta \in \mathbb{Z}_{\geq 0}$ (i.e., phase 1), from where the system reaches LS within $\delta \in \mathbb{Z}_{\geq 0}$ time units (i.e., phase 2).

Definition 12. *Let $\mathcal{SM} = (S, s_0, T)$ be the semantic model of a timed automaton with legitimate states LS, Q be a state predicate called* intermediate recovery predicate*, F be a set of faults, SPEC be a specification, and $\theta, \delta \in \mathbb{Z}_{\geq 0}$. The* strict 2-phase recovery *specification for $\mathcal{SM}$ is $SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq \theta} Q) \wedge (Q \mapsto_{\leq \delta} LS)$.* $\square$

[4] We note that while delay faults cannot be modeled explicitly due to the semantics of TADs, one can specify a delay fault by employing an additional location, where the delay occurs.

The other types of 2-phase recovery that are outside the scope of this paper are specified by different $SPEC_{\overline{br}}$ [7]. For example, ordered-strict recovery is specified by $SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq\theta} (Q - LS)) \wedge (Q \mapsto_{\leq\delta} LS)$. In order to define the notion of fault-tolerance using 2-phase recovery, we first characterize a notion where computations that can be produced in the presence of faults can be extended, such that they eventually meet the specification.

Definition 13. *A timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$* maintains *SPEC from state predicate SP iff*

- *SP is closed in T, and*
- *for every computation prefix $\overline{\alpha}$ of $\mathcal{SM}$ that starts in SP, there exists a computation suffix $\overline{\beta}$, such that $\overline{\alpha\beta} \in SPEC$.*

We say that TAD violates *SPEC from SP iff it is not the case that TAD* maintains *SPEC from SP.* □

Concerning Definitions 6 and 13, we note that if a timed automaton satisfies $SPEC$ from SP, then it maintains $SPEC$ from SP as well. However, the reverse direction does not always hold. Definition 13 is introduced for computations that TAD cannot produce, but can be extended to a computation in $SPEC$ by adding *recovery* computation suffixes.

Definition 14. *A timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$ is F-*tolerant *to SPEC from LS iff*

1. *$TAD \models_{LS}$ SPEC,*
2. *there exists an F-span FS of TAD from LS, such that*
   - *$(S, s_0, T \cup F)$ maintains SPEC from FS, where $SPEC_{\overline{br}}$ is as defined in Definition 12, and*
   - *$(S, s_0, T \cup F)$ satisfies FS $\mapsto_{<\infty}$ LS from FS.* □

The last condition is added to handle the case where response properties in $SPEC_{\overline{br}}$ are unbounded (since in this case, Definition 13 fails, as it only captures finite prefixes).

Example. Let Q be the set of states in which the automaton stays in waiting long enough to ensure that nothing bad happens; i.e., $Q \equiv (\mathsf{waiting} \wedge (x \geq 5))$. The timing-independent safety property for this automaton in defined in Subsection 2.2. The timing constraint of TAD is defined as follows:

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 6} Q) \;\wedge\; (Q \mapsto_{\leq 2} LS).$$

where the response times are chosen arbitrarily. $SPEC_{\overline{br}}$.

4 Problem Statement

Given are a fault-intolerant timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$ and legitimates states LS, a set F of faults, and specification $SPEC$, such that $TAD \models_{LS} SPEC$. Our goal is to develop an algorithm for synthesizing an automaton TAD' with semantic model $\mathcal{SM}' = (S', s_0, T')$ and legitimate states LS' from TAD, such that TAD' is F-tolerant to $SPEC$ from LS'. We require that the algorithm for adding fault tolerance does not introduce new behaviors to TAD in the absence of faults. To this end, we define the notion of *projection*. Intuitively, the projection of transitions T on state predicate SP includes all immediate transitions that start and end in SP, and the delay transitions that start in SP and remain in SP continuously.

Definition 15. *The* projection *of a set T of transitions on a state predicate SP is defined as follows:*

$$\begin{aligned} T \mid SP = \{&(s_0, s_1) \in T_{imm} \mid s_0, s_1 \in SP\} \cup \\ &\{(l, val_d, val_c) \to (l, val_d, val_c + \delta) \in T_d \mid \\ &\quad ((l, val_d, val_c) \in SP) \ \wedge \ (\forall \epsilon \in \mathbb{R}_{\geq 0} : ((\epsilon \leq \delta) \Rightarrow \\ &\qquad (l, val_d, val_c + \epsilon) \in SP))\} \end{aligned}$$ □

Using this definition, we clarify our requirement of not adding new behavior to TAD in the absence of faults. If LS' contains a state that is not included in LS, then TAD' may have a computation that reaches a state that is not reachable in TAD in the absence of faults. This may falsify $TAD' \models_{LS'} SPEC$ and, hence, we require $LS' \subseteq LS$. Likewise, if $T' \mid SP'$ contains a transition that is not included in $T \mid SP'$, then there may exist a computation in the synthesized model that is not in the original model in the absence of faults. Hence, we also require $(T' \mid SP') \subseteq (T \mid SP')$.

We assume there exists a clock for each bounded response property. This clock is needed to measure time when the first predicate in the property becomes true. Also, for simplicity and without loss of generality, we assume that when a fault occurs, no fault will happen until the system goes back to LS'. In [8], the authors present an algorithm based on region graphs that can deal with the case where faults can occur in the fault-span as well.

Problem statement. Given a fault-intolerant timed automaton TAD with semantic model $\mathcal{SM} = (S, s_0, T)$, a set F of faults, intermediate predicate Q, where $LS \subseteq Q$, and specification $SPEC$, such that $TAD \models_{LS} SPEC$, our goal is to propose an algorithm for synthesizing an automaton TAD' with $\mathcal{SM}' = (S', s'_0, T')$, and legitimate states LS' from TAD, such that:

1. $LS' \subseteq LS$,
2. Q is closed in T',
3. $(T' \mid LS') \subseteq (T \mid LS')$, and
4. TAD' is F-tolerant to $SPEC$ from LS'.

The constraint on closure of Q and $LS \subseteq Q$ are included, because otherwise the problem becomes NP-complete [7] in the size of time-abstract bisimulation of TAD. In this paper, our focus is on devising a zone-based algorithm for the case where the problem can be solved in polynomial time in the size of time-abstract bisimulation of TAD.

5 The Synthesis Algorithm

In this section, we present our zone-based algorithm for solving the problem of synthesizing a fault-tolerant TAD' from a given TAD as stated in Section 4.

5.1 Zone Graphs

Since the state space of a timed automaton is infinite, in order to formally analyze a timed automaton, we use an equivalent space-efficient finite symbolic transition system, called a *zone graph* [15]. Let χ be a set of *clock zones*, say ξ, inductively defined as $\xi ::= x \preceq n \mid x - y \preceq n \mid \xi \wedge \xi$, where x and y are clock variables, n is a constant integer, and $\preceq \in \{<, \leq\}$. Let ξ be a clock zone on the set of m clock variables and $[\![\xi]\!] = \{val_c \in \mathbb{R}^m_{\geq 0} \mid val_c \models \xi\}$. The operators *up* and *resdis* are defined on clock zones as follows:

- $up(\xi) = \{val_c + d \mid val_c \in [\![\xi]\!] \;\wedge\; d \in \mathbb{R}_{\geq 0}\}$
- $resdis(\xi, (X_{res}, X_{dis})) = \{val_c[X_{res}, X_{dis}] \mid val_c \in [\![\xi]\!]\}$

Observe that operator up has no effect on disabled clock variable. A *zone* z is a tuple $z = \langle l, val_d, [\![\xi]\!]\rangle$, where l is a location, val_d is a valuation of discrete variables, and $\xi \in \chi$ is a clock zone.

Definition 16. *Let $TAD = (L, l_0, V, U, X, E)$ be a timed automaton. The* zone graph *of TAD is defined as a transition system $\mathcal{Z}(TAD) = (Z, z_0, \rightsquigarrow)$, where*

- *Z is the set of zones defined on TAD*
- *$z_0 = \langle l_0, (val_d)_0, up(\overrightarrow{0}) \;\cap\; c(l_0, (val_d)_0)\rangle$*
- *$\rightsquigarrow$ is the relation defined on zones by: $\langle l, val_d, \xi\rangle \;\rightsquigarrow\; \langle l', val'_d, \xi'\rangle$, if there exists $(l, u, g_d, g_c, d, (X_{res}, X_{dis}), l') \in E$, such that $val_d \models g_d$, $u(val_d) = val'_d$, and $\xi' = up(resdis(\xi \wedge g_c, (X_{res}, X_{dis})) \;\cap\; c(l', val'_d)$.* $\square$

Example. Figure 2 shows the zone graph of the automaton in Figure 1.

We will use the following zone operators [5, 12] in our algorithm:

- $and(\xi_1, \xi_2)$ returns the conjunction of the constraints in ξ_1 and ξ_2.
- $down(\xi)$ returns the weakest precondition of ξ with respect to delay, which is the set of clock assignments that can reach ξ by some delay δ:

$$down(\xi) = \{val_c \mid val_c + \delta \in \xi \;\wedge\; \delta \in \mathbb{R}_{\geq 0}\}$$

- $free(\xi, x)$ removes all constraints on the clock x:

$$free(\xi, x) = \{val_c[x = \delta] \mid val_c \in \xi \;\wedge\; \delta \in \mathbb{R}_{\geq 0}\}$$

- $pred_e(\xi)$ computes the set of clock valuations that after some delay δ can take switch e, and reach ξ, and is formally defined as

$$pred_e(\xi) = \{val_c \mid (val_c + \delta) \models g_c \;\wedge (val_c + \delta)[X_{res}, X_{dis}] \in \xi \;\wedge \\ e = (l, u, g_d, g_c, d, (X_{res}, X_{dis})) \;\wedge\; \delta \in \mathbb{R}_{\geq 0}\}.$$

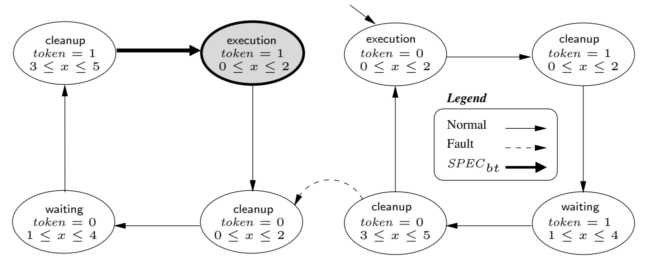
cleanup $token = 1$ $3 \leq x \leq 5$
execution $token = 1$ $0 \leq x \leq 2$
execution $token = 0$ $0 \leq x \leq 2$
cleanup $token = 1$ $0 \leq x \leq 2$
Legend
Normal
Fault
$SPEC_{bt}$
waiting $token = 0$ $1 \leq x \leq 4$
cleanup $token = 0$ $0 \leq x \leq 2$
cleanup $token = 0$ $3 \leq x \leq 5$
waiting $token = 1$ $1 \leq x \leq 4$

Fig. 2. Zone graph of the timed automaton in Figure 1

Algorithm 1 Zone_based_Synthesis

```
Input: A timed automaton $TAD$, with legitimate states $LS$, fault switches $F$, bad transitions $BT$, intermediate recovery
    predicate $Q$ st. $LS \subseteq Q$, recovery and intermediate recovery times $\delta$ and $\theta$.
Output: If successful, a fault-tolerant $TAD'$ with legitimate states $LS'$.
1: $TAD'' \leftarrow \mathsf{Enhance\_Automaton}(TAD, LS, F, Q, \delta, \theta, BT)$
2: $(Z, z_0, \rightsquigarrow) \rightarrow$ Construct_Zone_Graph $(TAD'')$
3: $(Z', z'_0, \rightsquigarrow), waiting \leftarrow \mathsf{Add\_Trans}((Z, z_0, \rightsquigarrow), BT)$
4: $(Z', z'_0, \rightsquigarrow) \leftarrow \mathsf{Backward\_Zones}((Z', z'_0, \rightsquigarrow'), \rightsquigarrow, waiting)$
5: $(Z', z'_0, \rightsquigarrow) \leftarrow$ Cycle_Removal $(Z', z'_0, \rightsquigarrow')$
6: $nz \leftarrow \{z_0 \mid \exists z_1, z_2 \ldots z_n \cdot (\forall j \mid 0 \leq j < n : (z_j, z_{j+1}) \in F'^z) \wedge (z_{n-1}, z_n) \in BT'^z\}$;
7: $Z_1 \leftarrow Z' - nz$
8: $LS_1^z \leftarrow LS'^z - nz$
9: $mz \leftarrow \{(z_0, z_1) \mid (z_1 \in nz) \vee (z_0, z_1) \in BT'^z\}$;
10: $\rightsquigarrow \leftarrow \rightsquigarrow - mz$
11: repeat
12:     $Z_2, LS_2^z \leftarrow Z_1, LS_1^z$;
13:     $nz \leftarrow \{z_0 \mid \nexists z_1 : (z_0, z_1) \in \rightsquigarrow\}$;
14:     $Z_1 \leftarrow Z_1 - nz$;
15:     $LS_1^z \leftarrow LS_1^z - nz$;
16:     $mz \leftarrow \{(z_0, z_1) \mid z_1 \in nz\}$;
17:     $\rightsquigarrow \leftarrow \rightsquigarrow - mz$;
18:     $nz' \leftarrow \{z_0 \mid (z_0, z_1) \in mz \cap F^z\}$;
19:     $Z_1 \leftarrow Z_1 - nz'$;
20:     $LS_1^z \leftarrow LS_1^z - nz'$;
21:     if $(Z_1 = \emptyset \vee LS_1^z = \emptyset)$ then
            print "no fault tolerant program exists";exit;
22:     end if
23: until $(Z_1 = Z_2 \ \wedge \ LS_1^z = LS_2^z)$
24: $TAD' \leftarrow Construct\_Automaton((Z_1, z_0, \rightsquigarrow), LS_1^z)$
25: return $TAD'$
```

5.2 Algorithm Sketch

Our zone-based algorithm consists in the following steps (Algorithm 1):

1. *(Automaton enhancement)* The input model is enhanced, so that the corresponding zone graph is more efficient and is augmented with delay transitions that can be utilized for adding 2-phase recovery.
2. *(Zone graph generation)* Next, the zone graph of the enhanced input automaton is generated. We utilize an existing algorithm from the literature of verification for this step.
3. *(Adding recovery behavior)* To enable 2-phase recovery, we add possible transitions among the zones of the zone graph. In this step, new zones may be added to the zone graph.
4. *(Backward zone generation)* For the newly added zones in the last step, we identify the backward reachable zones to ensure that the new zones do not introduce terminating computations.
5. *(Cycle removal)* Since adding recovery transitions may create cycles, the algorithm removes the possible cycles to ensure correct recovery.
6. *(Zone graph repair)* The zone graph is modified, so that it satisfies the safety properties in the presence of faults, and also does not introduce any deadlock states.

Finally, one can generate an automaton from the repaired zone graph. We consider this step as a black box, which gets a zone graph and returns a timed automaton corresponding to that semantic model.

Function 2 Enhance_Automaton

```
Input: A timed automaton $TAD = (L, l_0, V, U, X, E)$, with legitimate states $LS$, fault switches $F$, intermediate recovery
    predicate $Q$, recovery time $\delta$, intermediate recovery time $\theta$, and bad transitions $BT$
Output: An enhanced automaton $TAD' = (L', l_0, V, U, X', E')$

1: $X' \leftarrow X \cup \{x_f, x_q\}$
2: $L' \leftarrow L \cup \{\mathsf{deadlock}\}$
3: $E_0 \leftarrow \{(l, \mathfrak{u}, g_d, true, true, (\emptyset, X'), \mathsf{deadlock}) \mid \forall val_d \models g_d : (l, val_d) \in BS\}$
4: $E_1 \leftarrow \{(l, \mathfrak{u}, true, x_f = \theta, x_f = \theta, (\emptyset, X'), \mathsf{deadlock}) \mid l \in L\}$
5: $E_2 \leftarrow \{(l, \mathfrak{u}, true, x_q = \delta, x_q = \delta, (\emptyset, X'), \mathsf{deadlock}) \mid l \in L\}$
6: $F' \leftarrow \{(l_1, u, g_d, \phi, false, (r_1 \cup x_f, r_2), l_2) \mid \forall (l_1, u, g_d, \phi, false, (r_1, r_2), l_2) \in F\}$
7: $E_3 \leftarrow \{(l, u, g_d, \phi \wedge x_f \geq 0, true, (x_q, x_f), l) \mid \forall val_d \models g_d : \forall val_c \models \phi : (l, val_d, val_c) \in Q - LS\}$
8: $E_4 \leftarrow (l_1, u, g_d, g_c \wedge (x_f < 0), d, (r_1, r_2), l_2) \mid \forall (l_1, u, g_d, g_c, d, (r_1, r_2), l_2) \in E\}$
9: $E_5 \leftarrow (l_1, u, g_d, g_c \wedge (x_f \geq 0), false, (r_1, r_2), l_2) \mid \forall (l_1, u, g_d, g_c, d, (r_1, r_2), l_2) \in E\}$
10: $E' \leftarrow E \cup F' \cup \bigcup_{i=0}^{5} E_i$
11: return $TAD' = (L', l_0, V, U, X', E')$
```

5.3 Algorithm Description

The main algorithm (Algorithm 1) takes a timed automaton TAD, with legitimate states LS, fault transitions F, and intermediate recovery predicate Q st. $LS \subseteq Q$ as input. The specification consists of the time-independent safety specification (the set BT of bad transitions) and timing constraints (as the recovery time δ and intermediate recovery time θ).

Steps 1, 2: Automaton Enhancement / Zone Graph Generation Algorithm Zone_based_Synthesis starts by automaton invoking function Enhance_Automaton (see Function 2). The entire $\neg LS$ is (often) too large and impractical to build and explore. Hence, function Enhance_Automaton uses a heuristic to build a weak enough fault-span (rather than considering the entire $\neg LS$), such that we generate the zones only reachable using (1) the program switches, and (2) any possible delay, when the state of the model is in $\neg Q$. We exclude $Q - LS$, since adding delay transitions may violate the closure of Q. The clocks x_f and x_q are added to keep track of the time elapsed since a computation reaches $\neg LS$ and Q, respectively (Line 1). A new location, called deadlock (Line 2), along with the added switches leading to deadlock are used to prune the computations violating the specification.

The first set of pruned computations are those violating timing-independent safety specification in terms of bad states BS (Line 3). Computations reachable from a bad state can be pruned, and, hence, eager switches E_0 are used not to let time progress after we reach a bad state. The second set of states that can be used to prune the zone graph are the ones that violate timing constraints of 2-phase recovery:

- A computation cannot stay in $\neg LS - Q$ for more than θ time units. Hence, the set E_1 of switches are added to ensure that every computation that stays more than θ time units in $\neg LS - Q$ will be pruned (Line 4). Note that switches in E_1 are eager.
- Similarly, we respect the recovery time δ by adding the switches in E_2, which do not let time progress when the value of $x_q = \delta$ (Line 5).

Note that all added switches to the deadlock location disable all clocks. Also, a unique update function $\mathfrak{u}$ is used to set the value of discrete variables. This is done to avoid having multiple deadlock states with different clock valuations or discrete variables valuations in the semantic model.

Function 3 Add-Trans

Input: A zone graph $(Z, z_0, \rightsquigarrow)$, a set of legitimate zones LS^z, a set of intermediate recovery zones Q^z, a set of bad transitions BT^z
Output: A zone graph $(Z', z'_0, \rightsquigarrow')$, with recovery transitions being added, and a set of new subzones $waiting$

```
1: $waiting \leftarrow \emptyset$
2: $Z' \leftarrow Z$
3: $\rightsquigarrow' \leftarrow \rightsquigarrow$
4: FindZonesRanking $(Z, z_0, \rightsquigarrow)$
5: ConnectZones($Z - Q^z$, $Z$)
6: ConnectZones($Q^z - LS^z$, $Q^z$)
7: ConnectZonesRes($Z - Q^z$, $Z$)
8: ConnectZonesRes($Q^z - LS^z$, $Q^z$)
9: return $(Z', z'_0, \rightsquigarrow')$, $waiting$

10: function ConnectZones($Z_1$, $Z_2$: Set of zones){
11: for all $z \in Z_1, z' \in Z_2$ st. $(z, z') \notin (\rightsquigarrow \cup BT^z)$ do
12:     if $(rank(z) < \infty)$ break
13:     Let $z = (l, (val_d), \xi)$ and $z' = (l', (val'_d), \xi')$
14:     $\xi'' \leftarrow \xi \ \mathtt{to} \ \xi'$
15:     if $(\xi'' = \emptyset)$ continue
16:     $con(z) = 1$
17:     if $(\xi'' = \xi)$ then
18:         if $(rank(z) > rank(z') + 1)$ then
19:             $rank(z) = rank(z') + 1$
20:         end if
21:         $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup \{(z, z')\}$
22:     else
23:         $z'' \leftarrow (l, (val_d), \xi'')$
24:         $waiting \leftarrow waiting \cup \{(z'', z)\}$
25:         $Z' \leftarrow Z' \cup z''$
26:         $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup (z'', z')$
27:     end if
28: end for
    }

29: operator $\mathtt{to}(\xi_1, \xi_2$: Clock zone) {
30: for all $x \in X$ do
31:     if $upperbound(\xi_1, x) < lowerbound(\xi_2, x)$ then
32:         return $\emptyset$
33:     end if
34: end for
35: return $and\ (\xi_1, down(\xi_2))$
    }

36: function ConnectZonesRes($Z_1$, $Z_2$: Set of zones){
37: for all $z \in Z_1$ st. $\neg con(z) \wedge loc(z) \neq$ deadlock $\wedge$
    $\nexists z''' : (z, z''') \in \rightsquigarrow'$ , $z' \in Z_2$ st. $(z, z') \notin BT^z$ do
38:     if $(rank(z) < \infty)$ break
39:     Let $z = (l, (val_d), \xi)$ and $z' = (l', (val'_d), \xi')$
40:     $\xi'' \leftarrow \xi \ \mathfrak{tores} \ \xi'$
41:     if $(\xi'' = \emptyset)$ continue
42:     if $(\xi'' = \xi)$ then
43:         if $(rank(z) > rank(z') + 1)$ then
44:             $rank(z) = rank(z') + 1$
45:         end if
46:         $\rightsquigarrow' \leftarrow \rightsquigarrow \cup (z, z')$
47:     else
48:         $z'' \leftarrow (l, (val_d), \xi'')$
49:         $waiting \leftarrow waiting \cup \{(z'', z)\}$
50:         $Z' = Z' \cup z''$
51:         $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup (z'', z')$
52:     end if
53: end for
    }

54: operator $\mathfrak{tores}(\xi_1, \xi_2$: Clock zone, $X'$: Set of clocks) {
55: for all $x \in X'$ do
56:     if $lowerbound(\xi_2, x) \neq 0$ then
57:         return $\emptyset$
58:     end if
59: end for
60: for all $x \in X$ do
61:     if $x \notin X' \wedge upperbound(x, \xi_1) < lowerbound(x, \xi_2)$ then
62:         return $\emptyset$
63:     end if
64: end for
65: $\xi_3 = and\ (\xi_1, free(down(\xi_2), X'))$
66: return $\xi_3$
    }
```

The set F' of switches (Line 6) corresponds to the set F of faults, where the delay is set to $false$, as the fault transitions may not be taken in the computation, and with the clock c_f being added to the set of clocks to be reset. E_3 are eager switches that are triggered as soon as a state in $Q - LS$ is reached, where x_f is disabled and x_q is reset (Line 7). E_4 and E_5 are added, so that the switches of the program are lazy when the computation is not in Q, while they have the specified urgency when the computation in Q. This way, we allow any possible delay in $\neg Q$ for generating the fault-span (Lines 8 and 9).

Example. Figure 3 shows the result of applying our algorithm on the running example (Figure 1). The dashed zones are in $\neg LS$, and the dashed transitions corresponds to the fault. Zone 5 is generated by switch E_5. Adding this switch lets the states in $\neg LS - Q$ have any possible delay. Observe that lazy urgency of E_5 allows adding larger zones in $\neg Q$.

Step 3: Adding Recovery Paths After generating the enhanced automaton (Line 1), Algorithm 1 calls Function 3 (Add_Trans) to add recovery transitions (Line 3 of Algorithm 1). In order to reduce the complexity of this step, our idea is to first find the ranking of each zone in $\neg LS$ based on its possible path to LS, and then dynamically update this ranking during the recovery addition step. As soon as a zone in $\neg LS$ gets a ranking less than infinity (there is a path for it to LS), we stop finding a recovery transition starting from that zone. Adding recovery transitions in Function 3 is achieved by applying two strategies: (1) connecting existing zones to each other (Lines 5–6), and (2) connecting zones by resetting clocks for deadlock zones that cannot get connected using strategy 1 (Lines 7–8).

Strategy 1 After initializations (Lines 1-3 of Function 3), we add recovery transitions from zones in $\neg LS - Q$ to any possible zone, and also from zones in $Q - LS$ to any possible zone in Q (Lines 5 and 6, respectively) by calling function ConnectZones (defined in Lines 10–28). For adding the transitions between zones, one has to ensure that an added transition respects the clock constraints of source and target zones. To this end, we introduce the operator $\mathfrak{to}$ (defined in Lines 29-35) for finding the subset of a zone which can be connected to another zone. Two conditions for connecting two zones are:

– The upper bound of each clock variable in the first zone should be larger than its lower bound in the second zone. If this condition does not hold, then there is a time gap between the two zones.
– The time monotonicity condition should hold between them. For checking this condition, the intersection of the clock valuations that can reach the target zone, and the source zone is calculated. The result is a subzone of the source zone that can be connected to the target zone, which obviously can be empty or the original source zone.

If zone z is connected to zone z', we set the variable $con(z)$ to 1 to remember that a subset of this zone has been connected to another zone (Line 16). In case a new subzone z'' is created (Line 23), since ξ'' does not include all clock valuations of ξ, we need to ensure that all incoming computations to z'' respect time monotonicity. To this end, all new subzones are added to a waiting set (Line 24), which will be processed in Line 4 of Algorithm 1. Each member of the waiting list is a tuple with the first element being the new subzone, and the second being the original zone from which the subzone is formed.
Example. In Figure 3, zone 9 is added when Algorithm 3 attempts to connect zone 5 to zone 3 in strategy 1. Likewise, zone 6 is added when trying to connect zone 8 to zone 4. The transition from zone 7 to zone 1 is also added in this step.

Strategy 2 Next, Algorithm Add_Trans handles deadlock zones that could not be connected to other zones (Lines 7 and 8) by calling function ConnectZonesRes (defined in Lines 36–53). This strategy is identical to strategy 1, except it uses operator $\mathfrak{tores}$ (instead of $\mathfrak{to}$). This operator (defined in Lines 54–66) finds a subzone of the first zone that can be connected to the second zone by resetting a set of clock variables. Again applying this operator may result in creation of new subzones that are added to the set $waiting$ for later backward zone generation processing.

Step 4: Backward Zone Generation Since addition of recovery transitions in Step 3 may create new subzones (returned in $waiting$ by Function 3), if all incoming transitions

Function 4 Backward_Zones

Input: A zone graph $(Z', z'_0, \rightsquigarrow')$, the original set of transitions $\rightsquigarrow$, and a set of pairs of zones $waiting$.
Output: A zone graph $(Z', z'_0, \rightsquigarrow')$, with newly added zones being traced backward.

```
1: while $waiting \neq \emptyset$ do
2:     Let $(z_0, z_1)$ be a pair in $waiting$
3:     $waiting \leftarrow waiting - \{(z_0, z_1)\}$
4:     for all $z$ st. $(z, z_1) \in \rightsquigarrow$ do
5:         Let $e$ be the original switch for transition $(z, z_1)$
6:         Let $(l, (val_d), \xi) = z$ and $(l_1, (val_d)_1, \xi_1) = z_1$
7:         $\xi' \leftarrow pred_e(\xi_1)$
8:         $z' = (l, (val_d), \xi')$
9:         Let $waiting_0$ denote the set of first elements in $waiting$
10:        if $(z' \notin Z' \cup waiting_0)$ then
11:            $waiting = waiting \cup (z', z)$
12:        end if
13:        $\rightsquigarrow' = \rightsquigarrow' \cup (z', z_0)$
14:    end for
15: end while
16: return $(Z', z'_0, \rightsquigarrow')$
```

of the original superzone are added to the new subzone naively, we may introduce terminating computations. This happens when there are valuations in the predecessor zones of the original zone that cannot reach the new subzone. To address this case, Function 4 (Backward_Zones) is invoked for backward generation of predecessor zones for each new subzone in $waiting$ (called in Line 4 of Algorithm 1).

In Function 4, for each new zone in $waiting$, the switches (including faults) leading to the original zone are considered (Lines 4 and 5), and for each switch, the previous zone of the new zone using this switch is calculated using the $pred_e$ operator (Line 7). If the previous zone is not already included in the set of zones nor in the waiting list, it will be added to $waiting$ (Line 11). The algorithm repeats these steps until all backward reachable zones are explored and the appropriate transitions leading to the new zone are added (Line 13).

Example. In this step, zone 6 is traced backward using the switch corresponding to the transition from zone 5 to zone 6. The result is the added transition from zone 5 to zone 8. Zone 9 is likewise traced backward using the fault switch (corresponding to the transition from zone 4 to zone 5), and as a result, the transition from zone 4 to zone 9 is added.

Step 5: Removing Cycles Adding recovery transitions may lead to introducing a cycle in the zone graph, which violates the bounded response requirement. Thus, the possible added cycles are removed (Line 5 of Algorithm 1). Observe that our assumption on closure of Q will not allow any cycles to be formed between $Q - LS$ and $\neg LS - Q$. Hence, the only possibility of introducing a cycle is between zones in $\neg LS - Q$ and in $Q - LS$.

Removing the cycles can be implemented by applying classic graph-theoretic algorithms. We rank each zone in $\neg LS - Q$ (respectively, $Q - LS$) based on the length of the shortest path to a zone in Q (respectively, LS). For each transition in $\neg LS - Q$ (respectively, $Q - LS$), if the rank of the source is less than the rank of the target, then the transition will be removed, as it does not contribute in synthesizing a solution. This transition removal ensure cycle-freedom in the fault span.

Step 6: Zone Graph Repair In order to ensure that the synthesized zone graph does not violate timing-independent safety, in Lines 6–10, Algorithm 1 identifies and removes the set of zones/transitions from where faults alone can lead a computation to a state from

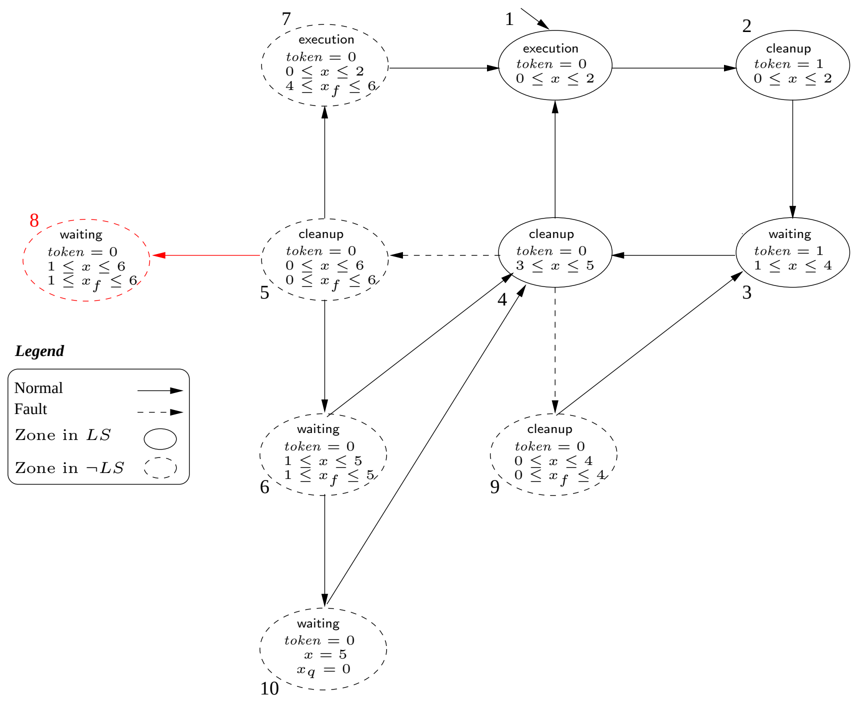

Fig. 3. Synthesized zone graph of the timed automaton in Figure 1

where safety can be violated (since occurrence of faults cannot be prevented). The rest of the algorithm (Lines 11–23 of Algorithm 1) removes deadlock zones and ensures the closure of legitimate states in the zone graph using a straightforward fixpoint computations. Finally, in Line 24 of Algorithm 1, it generates the output automaton out of the repaired zone graph.
Example. Zone 8 is a deadlock zone and, hence, gets removed in this step.

Theorem 1. *The algorithm Zone_based_Synthesis is sound.*

Proof. Refer to the appendix.

6 Implementation and Experimental Results

We have implemented our algorithm to evaluate the efficiency of our proposed synthesis method. We leveraged the IF toolset [11] for zone graph generation. IF provides an intermediate representation for specification of timed automata with urgency. It implements and evaluates different semantics of time, and various types of real-time constructs. We use the intermediate representation syntax to model a timed automaton with faults and automatically add switches to the the input model (Step 1 of Algorithm 1). Then we utilize the IF API to generate the zone graph of the enhanced automaton. The generated zone graph is stored in a graph data structure with zones being marked with LS, $Q - LS$, and $\neg Q$. Then, the rest of the algorithm (Steps 2 – 6) are performed on the generated zone graph. The result is a synthesized zone graph, which can be used to generate the fault-tolerant timed automaton. We tested our algorithm on two case studies.

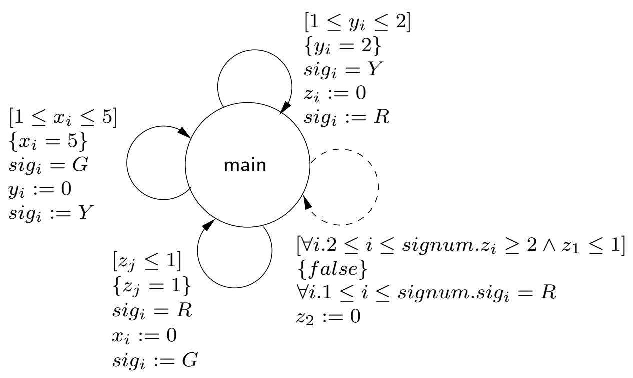

Fig. 4. Automaton for traffic controller

The first case study (adopted from [7]) is an automaton for a circular traffic controller (see Figure 4), with $signum$ signals. In this automaton, $j = (i+1) \mod 2$. The dashed switch shows the fault, and the others are switches of the input model. For each signal, a discrete variable sig_i is defined and ranges over $\{R, G, Y\}$. Also, there are three clock variables for each signal, x_i, y_i, and z_i, that act as timers to change the signal phase. All signals operate identically. One possible set of legitimate states for this model is the following predicate:

$$\begin{aligned} LS \equiv \forall i \in [0, signum).\ & [(sig_i = G) \Rightarrow ((sig_j = R) \wedge (x_i \leq 5) \wedge (z_i > 1))] \wedge \\ & [(sig_i = G) \Rightarrow ((sig_j = R) \wedge (y_i \leq 2) \wedge (z_i > 1))] \wedge \\ & [(sig_i = R) \wedge (sig_j = R) \Rightarrow ((z_i \leq 1) \oplus (z_j \leq 1))] \end{aligned}$$

, where $j = (i+1) \mod 2$, and $\oplus$ denotes an exclusive or operator. A bad transition is one that reaches a state where more than one signal is not red, which can be specified as follows:

$$bt = \{(s_0, s_1) \mid s_1 \models (\exists i, j.\ (i \neq j) \wedge (sig_i \neq R) \wedge (sig_j \neq R))\}$$

The fault (as can be seen in Figure 4) can reset z_2, when all z_is, except for z_1, are greater than 2. This fault can cause the system to reach a bad state. The bounded response property considered for this model is the following:

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 2} Q) \ \wedge \ (Q \mapsto_{\leq 3} LS)$$

Our second case study is a railway signal controller, consisting of $signum$ signals operating in a circular manner for controlling m trains (see Figure 5). In this automaton, $k = (i+1) \mod 2$. Train j is modeled by a discrete variable tr_j that ranges from 1 to $signum$, which shows the location of the train (i.e., the signal ahead of the train). When a train passes a signal, it changes phase from green and yellow to red. When a signal $i+2$ turns red, its previous signal $i+1$, which is also red, turns yellow. Then, if the previous signal i is yellow, it may turn green. It takes a train 5 time units to travel from one signal to the next. All signals operate similarly and, hence, the entire model of the train controller is the parallel composition of $signum$ timed automata illustrated in Figure 5.

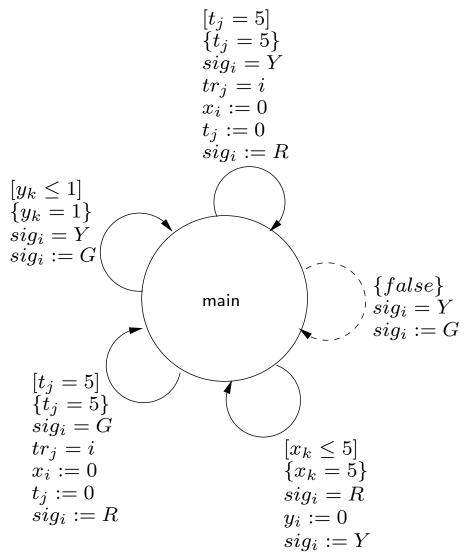

Fig. 5. Automaton for train signal controller

The safety specification of this model requires that no two train can have the same location at the same time, which can be represented by the following predicate for the bad transitions:

$$bt = \{(s_0, s_1) \mid s_1 \models (\exists i, j.\ (i \neq j) \wedge (tr_i = tr_j))\}$$

The fault in our case study occurs when the first signal changes phase from yellow to green due to circuit problems. This fault does not cause the computation to violate the specification, but it may result in a terminating computation, where trains cannot proceed due to the signal phases. The bounded response property considered for this model is the following:

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 2} Q)\ \wedge\ (Q \mapsto_{\leq 1} LS)$$

Table 1. Results for traffic signal controller

	3	**5**	**7**	**9**	**11**
Steps 1,2 (sec)	0.02	0.06	0.62	8.92	265.059
Steps 3-6 (sec)	0.02	0.02	0.07	0.10	0.15
Total synthesis time (sec)	0.04	0.08	0.69	9.02	265.209
Zone Graph Gen. Intolerant Model	0.0	$2h, 38m$	$> 3h$	$> 3h$	$> 3h$
Zone Graph Size Intolerant Model	309	1279032	$> 10^6$	$> 10^6$	$> 10^6$
Zone Graph Size of Enhanced Automaton	47	59	71	83	95

Table 2. Results for train signal controller

	4	**5**	**6**
Steps 1,2 (sec)	0.78	1.89	3.38
Step 3 (sec)	9.87	12.95	43.39
Step 4 (sec)	3.16	2.52	9.18
Step 5 (sec)	1.31	1.72	3.67
Step 6 (sec)	2.22	2.56	7.18
Total synthesis time (sec)	17.34	21.64	66.8
Zone Graph Gen. Intolerant Model	1.0	1.0	1.0
Zone Graph Size Intolerant Model	442	792	1112
Zone Graph Size of Enhanced Automaton	893	1424	1942

We compared the synthesis time to the zone graph generation time for the intolerant input automaton with fault (before enhancement) (Tables 1 and 2). This comparison enables us to analyze synthesis versus corresponding verification time.

In the traffic signal controller, as can be seen in Table 1, by increasing the model size, the zone graph generation time increases considerably, which turns out to be the bottleneck of our algorithm. However, this step outperforms the zone graph generation time for the original automaton with faults. This is because the fault leads to bad states, and a significant number of reachable zones are cut by our pruning switches added in Function 2.

Table 2 presents the results for the train signal controller. Each column shows the number of signals. In all experiments, there are two trains in the model. In this case study, as can be seen in Table 2, the bottleneck is mostly on the step for adding transitions among zones. The reason is due to the fact that in this model, the fault does not lead the computation to reach bad states and, hence, our pruning strategy does not help in this regard. Comparison between the number of zones in the original model and the enhanced one shows that there is an increase in the zone graph size. This is due to adding switches E_5 in Function 2, which let any possible delay in states out of Q. We should note that our idea for ranking the zones and updating the ranks dynamically has helped significantly to make this step more efficient. However, we believe using heuristics we can still make this step more efficient, although some of the solutions might be lost.

7 Related Work

The objective of fault diagnosis in timed automata [25] is to design a diagnoser which takes sequences of observable events (from a run of the automaton) as input and decides whether a fault has occurred. The announcement of a fault is made at most n steps after the fault occurrence.This work focuses on detecting faults, while our technique focuses on synthesizing a fault-tolerant timed automata.

Controller synthesis of timed systems is in spirit close to our work. Maler, et al. [22] propose an algorithm for synthesizing timed automata formulated by the notion of timed games. The idea is to define a predecessor function that finds the configurations from which the automaton can be forced to the desirable set of configurations, and the algorithm is a fixed-point iteration of this function. controller have the option of doing nothing and let the time pass, in addition to choosing among actions. We have made a similar decision to let the program have any possible delay when it is not in its legitimate state.

An on-the-fly algorithm on synthesizing timed models using zone graphs is proposed in [13], which is implemented in the tool UPPAAL-TIGA [4]. The algorithm is a symbolic extension of the on-the-fly algorithm suggested by Liu and Smolka [20]. The main idea of this work is (1) to use a combination of forward algorithm, and backward propagation, which helps the algorithm to terminate as soon as a winning strategy has been identified, and (2) to use zone graph as the underlying structure of the algorithm. We use similar ideas in the area of fault-recovery for timed models. The distinction of our work with these work (and also with [21]) is handling bounded response properties and, more importantly, adding recovery paths that the zone graph of the original model does not contain.

Automated addition of fault-tolerance to timed models has been studied with special focus on complexity analysis [7, 8]. To our knowledge, our work is the first in designing an efficient algorithm that can be used in practical tools with solid experimental results.

8 Conclusion

The goal of model synthesis is to generate computing artifacts that are correct by construction from existing models and logical specifications. Automated synthesis is known to be a notoriously difficult problem due to high complexity of the associated decision procedures. This complexity is further amplified in the context of timed formalisms that are widely used to model real-time embedded systems.

In this paper, we focused on synthesizing fault-tolerant timed models from their fault-intolerant version. The type of fault-tolerance under investigation is *strict 2-phase recovery*, where upon occurrence of faults, the system is expected to recover in two phases, each satisfying certain constraints. Our contribution is a synthesis algorithm that adds 2-phase strict fault recovery to a given timed model, while not adding new behaviors in the absence of faults. The latter is guaranteed due to the fact that the only transitions added to the semantic model are safe recovery transitions and our algorithm adds no transition that originates from a legitimate state of the input model.

The algorithm works on a space-efficient representation of timed models, know as the *zone graph*. To our knowledge, this is the first instance of such an algorithm. Our experiments show that the proposed algorithm can compete with model checking, where the synthesis time is proportional to the corresponding verification time (i.e., zone graph generation time for the input model).

For future work, we plan to investigate the instances of addition of phased recovery that are known to be NP-complete in the size of zone graph. Another research direction is to synthesize fault-tolerant timed models compositionally, where the input model is in terms of a set of interacting components.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Principles of Distributed Computing (PODC)*, pages 173–182, 1998.
4. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-Tiga: Time for playing games! In *CAV*, pages 121–125, 2007.
5. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.
6. R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification (CAV)*, pages 140–156, 2009.
7. B. Bonakdarpour and S. S. Kulkarni. Synthesizing bounded-time 2-phase recovery. In *Springer journal of Formal Aspects of Computing (FAOC)*. To appear.
8. B. Bonakdarpour and S. S. Kulkarni. Incremental synthesis of fault-tolerant real-time programs. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 122–136, 2006.
9. B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad. Symbolic synthesis of masking fault-tolerant programs. *Springer Journal on Distributed Computing (DC)*, 25(1):83–108, March 2012.
10. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *COMPOS*, pages 103–129, 1997.

11. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In *World Congress on Formal Methods*, pages 307–327, 1999.
12. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, pages 66–80, 2005.
13. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, pages 66–80, 2005.
14. P. Cerný, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *Computer Aided Verification (CAV)*, pages 243–259, 2011.
15. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International workshop on Automatic verification methods for finite state systems*, pages 197–212, 1990.
16. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
17. F. Faghih and B. Bonakdarpour. Zone-based synthesis of timed models with strict phased fault recovery. Technical Report CS-2013-04, University of Waterloo, 2013.
18. A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design (FMSD)*, 35(2):190–225, 2009.
19. T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43(3):135–141, 1992.
20. Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *ICALP*, pages 53–66, 1998.
21. O. Maler, D. Nickovic, and A. Pnueli. On synthesizing controllers from bounded-response properties. In *Computer Aided Verification (CAV)*, pages 95–107, 2007.
22. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 229–242, 1995.
23. P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
24. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM SIGPLAN Notices*, 41(11):404–415, 2006.
25. S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 205–224, 2002.

A Proof of Soundness (Theorem 1)

We show that any output of algorithm Zone_based_Synthesis meets the four conditions of the problem statement in Section 4. We distinguish four cases:

1. By construction, $LS' \subseteq LS$ trivially holds, as no state is added to LS. LS' might have some states removed compared to LS and those are the ones removed in Step 6. Also, observe that clock variables x_f and x_q are disabled in LS and, hence, their values are irrelevant in LS.
2. Q is closed in T'. Recall that Q is closed in the original model. The only switches we add to the automaton in Step1 originating from Q are the ones leading the states that do not satisfy the safety properties to the deadlock location. Note that the deadlock zone and all its incoming transitions will be removed in Step 6. Finally, in adding recovery transitions in Step 3, no transition is added from Q to $\neg Q$.
3. By construction, $(T' \mid LS') \subseteq (T \mid LS')$ also trivially holds, as no transition originating from LS is added.
4. TAD' is F-tolerant to $SPEC$ from LS'. To prove this condition, we distinguish two cases:
   - First, we have to show that $TAD' \models_{LS} SPEC$. By construction, and following cases 1 and 3, as well as the fact that the algorithm removes all deadlock states, it follows that the set of computations of TAD' is a subset of computations of TAD' in the absence of faults. Hence, we have $TAD' \models_{LS'} SPEC$.
   - We now need to show that there exists an F-span from where TAD' maintains $SPEC$ in the presence of faults. To this end, notice that if a computation reaches a state in $\neg LS$, by construction, no suffix of this computation includes a transition in BT. Hence, TAD' in the presence of faults maintains $SPEC_{\overline{bt}}$. Moreover, any computation that reaches a state in $\neg LS$ is guaranteed to reach Q and LS within θ and δ time units. This is ensured by Step 1 (by adding eager switches that do not let x_f and x_q exceed the allowed bounds), Step 4 (by not letting terminating computations being added to the synthesized model), Step 5 (by removing cycles), and Step 6 (by removing deadlocks and ensuring the closure of fault-span and LS). Observe that since all computations are guaranteed to reach LS, liveness is automatically preserved.