Symbolic Synthesis of Timed Models with Strict 2-Phase Fault Recovery

Fathiyeh Faghih
Department of Computing and Software, McMaster University, Canada
Email: faghihef@mcmaster.ca
Borzoo Bonakdarpour
Department of Computing and Software, McMaster University, Canada
Email: borzoo@mcmaster.ca

Abstract—In this article, 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. This is in part due to the fact that synthesis is in general a challenging problem and its complexity is significantly magnified in the context of timed systems. We propose an algorithm that takes as input a timed automaton, a set of fault actions, and a set of safety and bounded-time response properties, and utilizes a space-efficient symbolic representation of the timed automaton (called *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.

Keywords—synthesis, fault tolerance, real-time systems

1 Introduction

Dependability and *time-predictability* are two vital properties of most embedded (especially, safety/mission-critical) 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; fault-tolerance deals with unanticipated time and type of faults, while meeting timing constraints requires time predictability. This conflict 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 automated techniques that can generate fault-tolerant models that meet their timing constraints and are correct by construction.

Automated *synthesis* is a rigorous but highly complex method to generate models that are correct by construction. There are different synthesis paradigms, e.g., synthesis from a temporal logic specification [1], [2] along with quantitative objectives [3], [4], program sketching [5], and controller synthesis [6], [7]. Automated addition of fault-tolerance is also a technique for synthesizing a fault-tolerant model from its fault-intolerant version [8]–[10]. This line of work is in spirit close to controller synthesis, where faults can be modeled as uncontrollable transitions.

> *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, etc.), nature of the faults (permanent, transient, or intermittent), or the ability of the program to observe the effects of the faults [11]. We refer the reader to [8] for examples on different types of faults (e.g., fail-stop, Byzantine, state corruption, message loss, etc) modeled in a transition system.*

However, there are subtle differences (e.g., synthesis of recovery paths), which make the 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. In particular, fault-tolerance requires that the system should eventually return to its ideal behavior, and its real-time nature needs the recovery to be quick. While satisfying both requirements may not be possible, having two-phase fault recovery enables the system to first recover to a *safe* or acceptable state quickly, and then return to its *ideal* behavior. Also, the intermediate state could be useful for other purposes, e.g., for logging.

1.1 Motivating Example

Consider a one-lane bridge, where vehicles can travel only in one direction at each time instant. The bridge is controlled by two traffic signals, one at each end of the bridge using the set $\{x_1, x_2, y_1, y_2, z_1, z_2\}$ of clock variables. The controller performs the following set of

actions:

$$(sig_i = G) \wedge (1 \leq x_i \leq 10) \xrightarrow{y_i} sig_i := Y;$$
$$(sig_i = Y) \wedge (1 \leq y_i \leq 2) \xrightarrow{z_i} sig_i := R;$$
$$(sig_j = R) \wedge (z_i \leq 1) \xrightarrow{x_j} sig_j := G;$$

where $i \in \{0, 1\}$ and $j = (i + 1) \mod 2$. For instance, based on the first action, when the first signal is green, and the value of clock variable x_0 is between 1 and 10, clock variable y_0 gets reset and the signal turns yellow. One can observe that starting from a "*legitimate state*" (e.g., $s = (sig_0 = sig_1 = R) \wedge (z_0 \leq 1) \wedge (z_1 > 1)$), the controller works correctly, meaning that at each reachable state, at least one signal is red. Now, suppose that a fault action can reset variable z_1 due to a circuit malfunction. Starting from legitimate state s, if this fault occurs, both signals may go green based on the third action of the controller. This is clearly an unsafe state as vehicles can simultaneously enter the bridge on opposite directions.

Our goal in this article is that given such a fault-intolerant system, a set of faults, and a safety specification, we automatically synthesize a fault-tolerant system, such that: (1) the safety specification is never violated, and (2) when faults happen, the system goes back to its set of legitimate states in two phases within some specific time frame. For instance, in our example, when a fault occurs, both signals should first turn red immediately (an *acceptable* state, called Q) to prevent violation of safety specification (*i.e.,* phase 1) before final recovery to a legitimate state (*i.e.,* phase 2).

1.2 Contributions

In the context of synthesizing timed models with bounded-time phased fault recovery, let Q and LS be two predicates that should be reached in phases 1 and 2 of recovery within different time bounds, respectively. In [11], the authors showed that if Q is not required to be closed in the execution of recovery transitions, then synthesizing a timed automaton [12] with 2-phase recovery is NP-complete in the size of the detailed region graph [12] of the input automaton[1]. On the contrary, if the closure of Q is required and, moreover, $LS \subseteq Q$, then the synthesis problem can be solved in polynomial time. The polynomial-time algorithm presented in [11] to solve the latter problem is only a proof of concept to illustrate the problem complexity. In fact, the algorithm cannot be used in practice, since an implementation that utilizes a detailed region graph does not scale well. Intuitively, this is due to the fact that the size of region graphs grow exponentially in the size of timing constraints of the input timed model. As a matter of fact, even simple reachability (e.g., verification) problems in timed models are PSPACE-complete.

With this motivation, in this paper, we propose a time- and space-efficient algorithm that takes as input a timed automaton along with a set of fault transitions and synthesizes as output another timed automaton that provides 2-phase fault recovery, where Q is required to be closed and $LS \subseteq Q$. Our algorithm guarantees that no new behaviors are added to the input automaton in the absence of faults. This is guaranteed by adding no transitions that originate from states that can be reached only in the absence of faults. We note that our synthesis algorithm is sound, meaning that the synthesized fault-tolerant automaton satisfies all the requirements of 2-phase fault recovery. However, it is not complete, meaning that if it fails to find a fault-tolerant model for the given inputs, it is not guaranteed that there actually exists no solution for the given problem.

For space efficiency, we utilize *zone graphs* [13], developed as a symbolic finite representation of timed automata. Although there is work on synthesis of timed models using zone graphs (most notably the tool UPPAAL-Tiga [14]), the state of the art currently lacks two sets of techniques to make synthesis of fault-tolerant timed models possible:

- zone-based addition of safe recovery paths that do not exist in the original intolerant model, and
- efficient zone-based controller synthesis for bounded response properties of the form $Q \mapsto_{\leq \delta} LS$; *i.e.,* when Q becomes true, LS should become true within δ time units.

These are challenging problems, as adding or splitting zones during synthesis without considering their properties may lead to generating deadlock computations or timing gaps.

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 [15]. 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. We emphasize that although it is possible to synthesize controllers for bounded response specifications in UPPAAL-Tiga, in order to synthesize bounded-time fault recovery paths using UPPAAL-Tiga, one has to augment the input model with possible recovery transitions, introduce new clock variables to keep track of time elapsed for each phase of recovery, and make optimizations to make synthesis practical. These are essentially the ingredients of our technique. 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), although synthesis is a substantially more complex problem.

1. A detailed region graph is a finite bisimilar representation of a timed automaton. The size of region graph is exponential in the size of clock constraints. For example, the region graph of the automaton in Fig. 1 has 36 regions, compared to 8 zones in the corresponding zone graph (Fig. 3).

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 problem, while Section 5 presents our zone-based synthesis algorithm. We describe our implementation, case studies, and experimental results in Section 6. Related work and concluding remarks are discussed in Sections 7 and 8, respectively.

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)

In this paper, we adopt the notion of timed automata [12] with deadlines (TAD) [16] extended by discrete variables.

2.1.1 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* [2] 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 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* (Boolean expressions) over V by G_D.

Definition 1: A *timed automaton with deadlines* 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*.

A timed switch is of the form $(l, u, g_d, g_c, ur, (X_{res}, X_{dis}), l')$, 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 $ur \in \Phi$ is the transition urgency, such that $ur \Rightarrow g_c$, where '$\Rightarrow$' denotes logical implication. □

In Definition 1, ur determines the urgency of a switch. There are three different types of urgencies [16]:

- *(Eager)* When $ur = g_c$, the switch is called *eager*. An enabled eager switch cannot be delayed and, hence, does not let time progress before its execution.
- *(Lazy)* If $ur = false$, 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.
- *Delayable* In a *delayable* switch, ur 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.

2. Time does not advance for a disabled clock until it gets reset. We need disabled clocks to measure the time that the system is out of its set of legitimate states, as will be discussed in Section 5.

2.1.2 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 $\mathcal{SM} = (S, s_{init}, T)$, 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.
- $s_{init} = (l_0, (val_d)_0, \overrightarrow{0})$ 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:

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

and is called the *time progress condition* of location l and valuation val_d. Note that if each of the clauses in the disjunction gets true, the time cannot progress before switch e is taken, after which the system will no longer be in a state, where location is l and the set of valuations is 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:** $(l, val_d, val_c) \rightarrow (l', val'_d, val_c[X_{res}, X_{dis}])$ is an immediate transition iff there exists a switch $(l, u, g_d, g_c, ur, (X_{res}, X_{dis}), l') \in E$, 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 is denoted by T_{imm}.

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

2.1.3 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, one of the processes is the master process (illustrated in Fig. 1)[3]. 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 urgency is identified by {}.

The master process stays in execution for one to two time units. Then, it resets x, toggles the value of $token$, and goes to cleanup, where it can spend another one to two 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.

Definition 2: A *state predicate SP* of a semantic model $\mathcal{SM} = (S, s_{init}, T)$ 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 $\mathcal{SM} = (S, s_{init}, T)$ 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 $\tau_j, \tau_{j+1}, \ldots$ (called the *global time*), satisfies the following conditions:

3. The automaton for the slave process is not shown here for space reasons, but one can consider an automaton of two states, execution and waiting, with two switches between them. The condition of the transitions is the value of $token$.

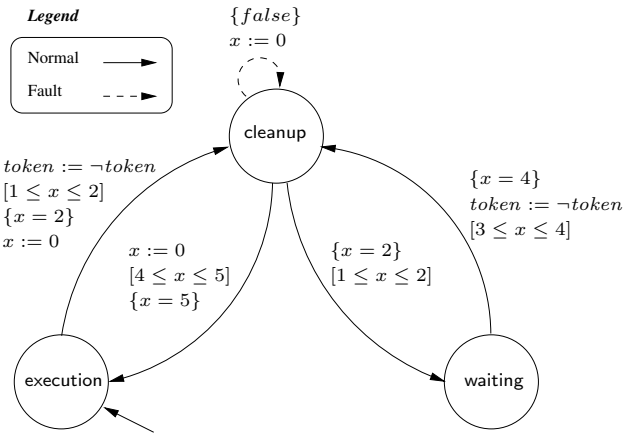

Fig. 1. A timed automaton with deadline augmented with one fault switch

– *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 $\tau_{i+1} = \tau_i$. □

Definition 4: A *specification* is a set of infinite computations that satisfy time-monotonicity and divergence [17]. □

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 $\mathcal{SM} = (S, s_{init}, T)$ 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 (in Section 3), 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, $SPEC$ be a specification, and LS be a nonempty state predicate of TAD. We say that LS is a set of *legitimate states* of TAD for $SPEC$ iff $TAD \models_{LS} SPEC$. □

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 $\overline{s} = (s_0, \tau_0) \rightarrow (s_1, \tau_1) \rightarrow \ldots$, where for all $i \geq 0$, if $s_i \in P$, then there exists $k \geq i$, such that $s_k \in Q$ and $\tau_k - \tau_i \leq \delta$. □

In this paper, our notion of specification consists of two parts: (1) a *safety specification*, and (2) a *liveness specification* [17], [18]. 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*[4] bt. The specification in which each computation has no bad transitions is denoted by $SPEC_{\overline{bt}}$.
2) *Timing Constraints*: Denoted by $SPEC_{\overline{br}}$ is the conjunction of m bounded-time response properties: $\bigwedge_{i=1}^{m}(P_i \mapsto_{\leq \delta_i} Q_i)$. □

A bad transition that can be specified by its target state only defines a set of *bad states*. In the context of our example, a state in which $token = 1$ and the model is in location execution is a bad state. Note that it is not always the case that bad transitions can be identified by bad states. For example, in a traffic signal controller, a bad transition can be a transition originating from a state where the signal is initially red and turns yellow in the target state.

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 $\overline{\alpha}\overline{\beta} \in SPEC$. □

Following [18] and [17], any specification is an intersection of some safety and some liveness specifications, and hence, we don't repeat liveness in the specification representation.

2.2.1 Example

Consider the timed automaton in Fig. 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:

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

It is straightforward to see that starting from any state in LS, execution of *normal* switches of the automaton in Fig. 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 [11].

4. Execution of a bad transition leads to violation of the safety specification.

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 [11].

Given a semantic model $\mathcal{SM} = (S, s_{init}, T)$, a set F of faults is a subset of all possible immediate transitions[5]. In other words, $F \subseteq (S \times S)_{imm}$, where

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

We required that $X_{dis} = \emptyset$, because we assume faults cannot disable any clocks.

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 $\mathcal{SM} = (S, s_{init}, T)$, 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 $T \cup F$. □

Hence, a fault-span is a state predicate up to which (but not beyond which) faults can perturb the state of a system. In other words, fault-span is the set of all states reachable by faults and program transitions. 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.

3.1.1 Example

In Fig. 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 urgency 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 $token = 0$, and then 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 fault recovery* [11], when the state of a system is perturbed by a fault, the system is

5. 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.

required to either directly return to its legitimate states LS within $\theta \in \mathbb{N}$ time units, or, if direct recovery is not feasible, then it should first reach an *intermediate* recovery predicate Q within $\theta \in \mathbb{N}$ (*i.e.,* phase 1), from where the system reaches LS within $\delta \in \mathbb{N}$ time units (*i.e.,* phase 2).

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

Throughout this paper, we consider $SPEC_{\overline{br}}$, which is part of the specification, to be defined as above. The other types of 2-phase recovery that are outside the scope of this paper are specified by different $SPEC_{\overline{br}}$ [11]. 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 $\mathcal{SM} = (S, s_{init}, T)$ *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. □

Considering 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: An automaton TAD with semantic model $\mathcal{SM} = (S, s_{init}, T)$ is *F-tolerant* to $SPEC$ from LS iff

1) $TAD \models_{LS} SPEC$,
2) there exists an F-span FS of TAD from LS, st.
   - $(S, s_{init}, T \cup F)$ maintains $SPEC$ from FS,
   - $(S, s_{init}, T \cup F)$ 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).

3.2.1 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.,* $Q \equiv (\text{waiting} \wedge (x \geq 5))$. The timing-independent safety property for this automaton is defined in Subsection 2.2. The timing constraint 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. The property $SPEC$ is the conjunction of $SPEC_{\overline{bt}}$ and $SPEC_{\overline{br}}$.

4 Problem Statement

Given are a fault-intolerant timed automaton TAD with semantic model $\mathcal{SM} = (S, s_{init}, T)$ and legitimate states LS for a specification $SPEC$ (i.e., $TAD \models_{LS} SPEC$), and a set F of faults. Our goal is to develop an algorithm for synthesizing an automaton TAD' with semantic model $\mathcal{SM}' = (S', s_{init}, T')$ 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:

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

Recall that T_{imm} and T_d are the sets of immediate and delay transitions in T, respectively. 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 $T' \mid LS'$ contains a transition that is not included in $T \mid LS'$, 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 $(T' \mid LS') \subseteq (T \mid LS')$.

We assume there exists a clock variable for each bounded response property. This clock is needed to measure the time elapsed since the first predicate in the property becomes true. Also, for simplicity and without loss of generality, we assume when a fault occurs, no other fault happens until the system goes back to LS'. In [9], the authors present an algorithm based on region graph that can deal with the case where faults occur in the fault-span as well.

Problem statement. Given a fault-intolerant timed automaton TAD with semantic model $\mathcal{SM} = (S, s_{init}, T)$, its set LS of legitimate states for a specification $SPEC$ (i.e, $TAD \models_{LS} SPEC$), a set F of faults, and intermediate predicate Q, where $LS \subseteq Q$, our goal is to propose an algorithm for synthesizing an automaton TAD' with $\mathcal{SM}' = (S', s'_{init}, T')$, 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 [11] 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. Througout the paper, we refer to the input program as the *fault-intolerant* timed automaton.

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* [13]. A *clock zone* ξ is 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 $\preceq \in \{<, \leq\}$. As an example, Fig. 2 represents the clock valuations encoded in the clock zone ξ, where we have two clock variables x and y.

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:

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

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 ξ 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 $(l, u, g_d, g_c, ur, (X_{res}, X_{dis}), l') \in E$, such that $val_d \models g_d$, $u(val_d) = val'_d$, and $\xi' = up(resdis(\xi \wedge g_c, (X_{res}, X_{dis})) \cap c(l', val'_d)$. □

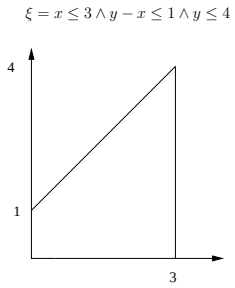

Fig. 2. An example of a zone

For example, Fig. 3 shows the zone graph of the automaton in Fig. 1.

We use the following zone operators [19], [20] 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

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

5.2 Algorithm Sketch

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

1) *Automaton enhancement:* The input model is enhanced with a new location ("sink"), and a number of switches (depending on the number of states violating the safety specification as explained in Section 5.3.1) entering it, which prune computations that violate the given specification. As a result, the corresponding zone graph will be more space-efficient. Also, the model is augmented with two clocks, and delay transitions that can be utilized for adding 2-phase recovery within specific delays.
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 [15] for this step.
3) *Adding recovery behavior:* To enable 2-phase recovery, we add some of the possible transitions among

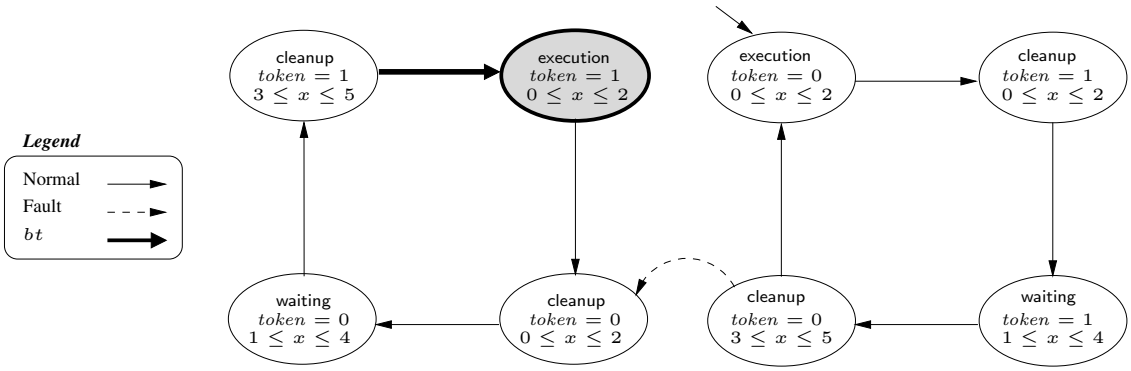

Fig. 3. Zone graph of the timed automaton in Fig. 1

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 specification in the presence of faults, and also does not contain 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.

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 such that $LS \subseteq Q$ as input. The specification consists of the timing-independent safety specification (the set BT of bad transitions) and timing constraints (as the recovery time δ and intermediate recovery time θ).

5.3.1 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 sink (Line 2), along with the added switches leading to sink 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).

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 δ and θ.
Output: If successful, a fault-tolerant TAD' with legitimate states LS'.

1: $TAD'' \leftarrow$ Enhance_Automaton$(TAD, LS, F, Q, \delta, \theta, BT)$
2: $(Z, z_0, \rightsquigarrow) \rightarrow$ Construct_Zone_Graph (TAD'')
3: $(Z', z_0', \rightsquigarrow), waiting \leftarrow$ Add_Trans$((Z, z_0, \rightsquigarrow), BT)$
4: $(Z', z_0', \rightsquigarrow) \leftarrow$ 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 found";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'

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 δ, intermediate recovery time θ, 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: $L' \leftarrow L \cup \{\mathsf{sink}\}$
3: $E_0 \leftarrow \{(l, \mathfrak{u}, g_d, true, true, (\emptyset, X'), \mathsf{sink}) \mid \forall val_d \models g_d : (l, val_d) \in BS\}$
4: $E_1 \leftarrow \{(l, \mathfrak{u}, true, x_f = \theta, x_f = \theta, (\emptyset, X'), \mathsf{sink}) \mid l \in L\}$
5: $E_2 \leftarrow \{(l, \mathfrak{u}, true, x_q = \delta, x_q = \delta, (\emptyset, X'), \mathsf{sink}) \mid l \in L\}$
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: $E_4 \leftarrow (l_1, u, g_d, g_c \wedge (x_f < 0), ur, (r_1, r_2), l_2) \mid \forall (l_1, u, g_d, g_c, ur, (r_1, r_2), l_2) \in E\}$
9: $E_5 \leftarrow (l_1, u, g_d, g_c \wedge (x_f \geq 0), false, (r_1, r_2), l_2) \mid \forall (l_1, u, g_d, g_c, ur, (r_1, r_2), l_2) \in E\}$
10: $E' \leftarrow E \cup F' \cup \bigcup_{i=0}^{5} E_i$
11: **return** $TAD' = (L', l_0, V, U, X', E')$

Note that all added switches to the sink 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 sink states with different clock valuations or discrete variables valuations in the semantic model.

The set F' of switches (Line 6) corresponds to the set F of faults, where the urgency 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 is in Q. This way, we allow any possible delay in $\neg Q$ for generating the weak enough fault-span (Lines 8 and 9).

5.3.2 Example

Fig. 4 shows the result of applying our algorithm on the running example (Fig. 1). The dashed zones are in $\neg LS$, and the black thin dashed transitions correspond 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. Zone 11 is the "sink" zone, which is used to prune computations leading to violate the specification.

5.3.3 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 $\neg LS - Q$ (respectively, $Q - LS$) based on the length of the shortest path to a zone in Q (respectively, LS), and then dynamically update this ranking during the recovery addition step. As soon as the ranking of a zone in $\neg LS$ is less than infinity (there is a path for it to LS), we stop finding a recovery transition 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).

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: $\xi'' \leftarrow \mathfrak{to}(\xi, \xi')$
15: **if** $(\xi'' = \emptyset)$ **continue**
16: $con(z) = 1$
17: **if** $(\xi'' = \xi)$ **then**
18: $rank(z) = rank(z') + 1$
19: $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup \{(z, z')\}$
20: **else**
21: $z'' \leftarrow (l, (val_d), \xi'')$
22: $waiting \leftarrow waiting \cup \{(z'', z)\}$
23: $Z' \leftarrow Z' \cup z''$
24: $\rightsquigarrow' \leftarrow \rightsquigarrow' \cup (z'', z')$
25: **end if**
26: **end for**}

27: **operator** $\mathfrak{to}(\xi_1, \xi_2$: Clock zone) {
28: **for all** $x \in X$ **do**
29: **if** $ub(\xi_1, x) < lb(\xi_2, x)$ **then**
30: **return** $\emptyset$
31: **end if**
32: **end for**
33: **return** $and\,(\xi_1, down(\xi_2))$ }

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

50: **operator** $\mathfrak{tores}(\xi_1, \xi_2$: Clock zone) {
51: Let $X' = \emptyset$
52: **for all** $x \in X$ **do**
53: **if** $lb(\xi_2, x) = 0$ **then**
54: $X' = X' \cup \{x\}$
55: **end if**
56: **end for**
57: **for all** $x \in X$ **do**
58: **if** $x \notin X' \wedge ub(x, \xi_1) < lb(x, \xi_2)$ **then**
59: **return** $\emptyset$
60: **end if**
61: **end for**
62: $\xi_3 = and\,(\xi_1, free(down(\xi_2), X'))$
63: **return** ξ_3
64: }

- **Strategy 1.** After initializations (Lines 1-3 of Function 3), we add recovery transitions from zones

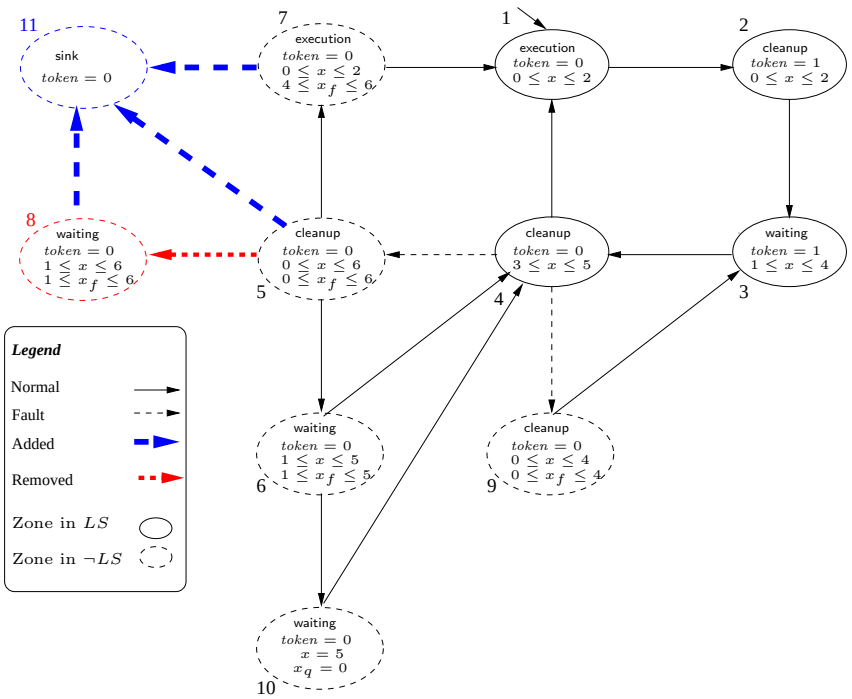

Fig. 4. Synthesized zone graph of the timed automaton in Fig. 1

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 10–26). 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 to (defined in Lines 27-33) 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 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 21), 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 22), 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.

- **Strategy 2.** Next, Function Add_Trans handles deadlock zones that could not be connected to other zones (Lines 7 and 8 of Function 3) by calling function ConnectZonesRes (Lines 34–49). This strategy is identical to strategy 1, except it uses operator tores (instead of to). This operator (defined in Lines 50–64) finds a subzone of the first zone that can be connected to the second zone by resetting a set of clock variables[6]. Again applying this operator may result in creation of new subzones that are added to the set *waiting* for later backward zone generation processing.

5.3.4 Example

In Fig. 4, zone 9 is added when Function 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.

5.3.5 Step 4: Backward Zone Generation

Addition of recovery transitions in Step 3 may create new subzones (returned in *waiting* by Function 3). If all incoming transitions 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. As an example,

6. The clocks that their lower bounds are 0 in the target zone can be reset.

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** $(Z', z'_0, \rightsquigarrow')$

consider three zones $\xi_1 ::= 1 \leq x \leq 5$, $\xi_2 ::= 3 \leq x \leq 5$, and $\xi_3 ::= 1 \leq x \leq 4$. There is a transition from ξ_1 to ξ_2. Assume that in Step 3, we tried to add a transition from ξ_2 to ξ_3, and as a result, a new zone $\xi'_2 ::= 3 \leq x \leq 4$ is generated, and the transition (ξ'_2,ξ_3) is added. Now, if we add a transition from the predecessor of ξ_2, which is ξ_1, to ξ'_2, a terminating computation is generated. The problem arises when the computation is in ξ_1, and the clock x has the value $4 < x \leq 5$; it cannot take the added transition to get to the zone $\xi'_2 ::= 3 \leq x \leq 4$, since the clock value should decrease to make this happen. 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). Function 4 repeats these steps until all backward reachable zones are explored and the appropriate transitions leading to the new zone are added (Line 13).

5.3.6 Example

In this step, zone 6 is traced backward using the switch corresponding to the transition from zone 5 to zone 8. The result is the added transition from zone 5 to zone 6. 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.

5.3.7 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 possibilities of introducing a cycle are between zones in $\neg LS - Q$ and in $Q - LS$.

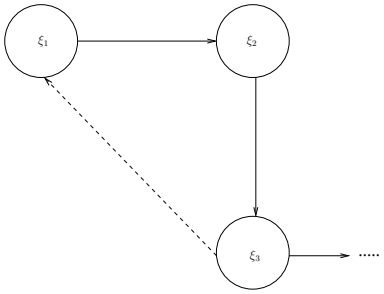

Fig. 5. An example of cycle removal

Removing the cycles can be implemented by applying classic graph-theoretic algorithms. Note that we have the rank of 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 ensures cycle-freedom in the fault span. As an example, consider a part of a zone graph shown in Fig. 5. Assume that all three zones are in $\neg LS$, and as shown, the recovery paths of all three zones ξ_1, ξ, and ξ_3 are through the outgoing path from ξ_3. It is obvious that the rank of ξ_3 is less than the rank of ξ_1, and hence, the transition between them will be removed. By removing this transition, the cycle among these three zones will be removed as well.

5.3.8 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 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 computation. Finally, in Line 24 of Algorithm 1, it generates the output automaton out of the repaired zone graph.

5.3.9 Example

Zones 8 and 11 are deadlock zones and, hence, get removed in this step. The automaton in Fig. 6 can be generated out of the repaired zone graph in Fig. 4.

5.3.10 Correctness of the Algorithm

Theorem 1: Zone_based_Synthesis algorithm is sound.

Proof. We show that any output of algorithm Zone_based_Synthesis is sound. In other words, it meets the four conditions of the problem statement in Section 4. We distinguish four cases:

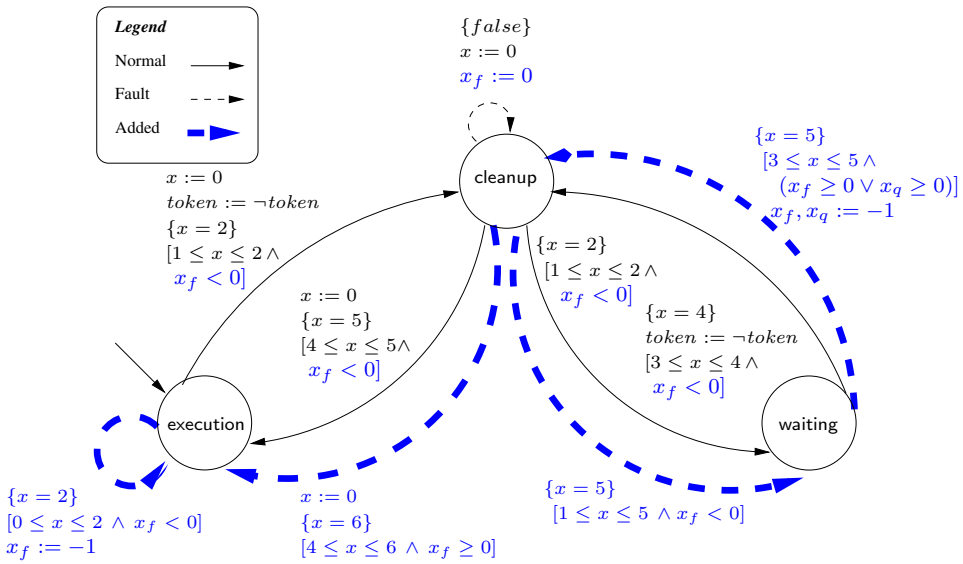

Fig. 6. Repaired automaton of the TAD in Fig. 1

1) By construction, $LS' \subseteq LS$ trivially holds, as no state is added to LS. LS' may 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 Step 1 originating from Q are the ones leading the states that do not satisfy the safety properties to the sink location. Note that the sink 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. □

Theorem 2: Zone_based_Synthesis algorithm is terminating.

Proof. Now, we show that algorithm Zone_based_Synthesis is terminating. Steps 1 and 2 are clearly terminating, as automaton enhancement has no loops and zone graph generation for a finite set of locations is always guaranteed to terminate [19].

In step 3, zones are first ranked based on their shortest paths to LS. This is done by a slightly modified version of Dijkstra's shortest path algorithm. Then, recovery transitions are added among the zones, which is, in the worst case, quadratic in the size of the zone graph, and hence, terminating. In the next step, zones reachable backward from the newly added zones are calculated. Since only a finite number of backward reachable zones can be generated, this step is also terminating.

The cycle removal step is done by checking the ranks of the source and target zones of transitions in $\neg LS$, and hence, in the worst case, is quadratic in the size of the zone graph. In the last step, the zone graph is repaired by removing the bad states and deadlocks. Deadlock removal is done in a loop, which terminates when a fixpoint is reached, or all zones are removed. It will always reach a fixpoint (if the zone graph does not get empty), since in each iteration the deadlock zones and their incoming transitions are removed. Transition

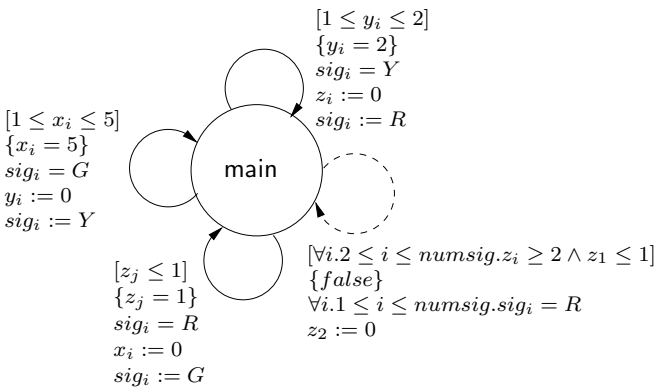

Fig. 7. Automaton for traffic controller i

removal may make more zones deadlock, which will be removed in the next iteration. If no new deadlock is formed in an iteration, a fixpoint is reached, and the loop terminates. Since the zone graph is finite, it will eventually reach a fixpoint, or the zone graph gets empty. □

6 Implementation and Experimental Results

We have implemented our algorithm to evaluate the efficiency of our synthesis method. We leveraged the IF toolset [15] 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) is performed on the generated zone graph. The result is a synthesized zone graph, which can be used to generate the fault-tolerant timed automaton. To evaluate our algorithm, we conducted three case studies.

6.1 Case Study 1: Circular Traffic Controller

The first case study (adopted from [11]) is an automaton for a circular traffic controller (Fig. 7), with $numsig$ number of signals. In this automaton, $j = (i+1)\ mod\ 2$. The dashed switch is the fault and solid switches are the ones given in the input model. For each signal, a discrete variable sig_i 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. For instance, when a signal i is green, it will turn yellow at least after one time unit and at most within 5 time units (*i.e.*, $1 \leq x_i \leq 5$). Such change of phase resets clock y_i, which keeps track of the time elapsed since signal i has turned yellow. All signals operate identically. One possible set of legitimate states for this model is the following predicate:

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

where $j = (i+1)\ mod\ numsig$, and $\oplus$ denotes the exclusive-or operator. A bad transition is one that reaches a state where more than one signal is not red:

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

The fault (as can be seen in Fig. 7) can reset (for instance) z_2, when all z-clocks, except for z_1, are greater than 2, due to a circuit malfunction. We consider the following bounded response property for this model:

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

where $Q = \forall i \in [0, numsig).(sig_i = R) \wedge (z_i \geq 2)$. Table 1 shows the breakdown of the time spent in different steps of our synthesis algorithm in the first three rows for 3-11 traffic signals. As expected, synthesis time increases as we increase the number of traffic signals. However, observe that the bottleneck of our algorithm turns out to be zone graph generation and not the synthesis steps. Thus, to better evaluate our algorithm, we compare it with its corresponding verification time. Notice that zone graph generation of the enhanced automaton (first row) significantly outperforms zone graph generation time for the original automaton with faults (fourth row). This is because the fault leads to bad states and a significant number of reachable zones are eliminated by our pruning switches added in Function 2, and hence, the number of zones in the zone graph of the enhanced automaton is less than the original zone graph. Having a smaller zone graph also assists in increasing the efficiency of the next steps of our algorithm.

TABLE 1
Results for traffic controller of 3-11 signals

	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 Generation of Intolerant Model	0.0 s	$2h, 38m$	$> 3h$	$> 3h$	$> 3h$
Zone Graph Size of Enhanced Automaton	47	59	71	83	95
Zone Graph Size of Intolerant Model	309	1279032	$> 10^6$	$> 10^6$	$> 10^6$

6.2 Case Study 2: Train Signal Controller

Our second case study is a railway signal controller, consisting of $numsig$ signals operating in a circular manner for controlling m trains (Fig. 8). In this automaton, $k = (i+1) \mod 2$. Train j is modeled by a discrete variable tr_j that ranges from 1 to $numsig$, which shows

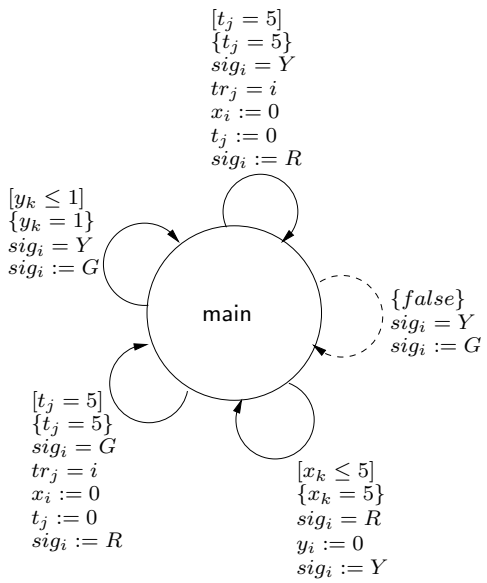

Fig. 8. Automaton for train signal controller

the location of the train (*i.e.,* the signal ahead of the train). When a train passes a signal, it changes phase from green or 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 identically, and hence, the entire model of the train controller is the parallel composition of $numsig$ timed automata illustrated in Fig. 8.

The safety specification of this model requires that no two trains can be in the same location at the same time:

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

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 deadlock computation, where trains cannot proceed due to deadlocked signals. The bounded response property considered for this model is the following:

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

where $Q = \forall i \in [0, numsig).(sig_i = R) \wedge (z_i \geq 6)$. One possible set of legitimate states is the set of states reachable from the initial state, where no two trains are in the same location, by the switches of the timed automaton (Fig. 8).

Table 2 presents the results for 4-6 signals and constant number of two trains. As can be seen, the bottleneck is mostly in the step for adding transitions among zones. This 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 is not necessarily helpful. 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 note that our idea for ranking the zones and updating the ranks dynamically has significantly made this step more efficient. However, we believe that using heuristics, we can still make this step more efficient at the cost of losing completeness.

TABLE 2
Results for train signal controller

	4	5	6
Steps 1,2 (sec)	0.42	1.01	3.01
Step 3 (sec)	2.1	2.3	47.7
Step 4 (sec)	0.0	0.5	7.98
Step 5 (sec)	0.31	0.37	3.87
Step 6 (sec)	0.0	0.0	7.23
Total synthesis time (sec)	3.68	4.18	69.79
Zone Graph Generation of Intolerant Model	1.0 s	1.0 s	1.0 s
Zone Graph Size of Enhanced Automaton	597	995	1942
Zone Graph Size of Intolerant Model	442	792	1112

6.3 Case Study 3: Fischer's Mutual Exclusion Algorithm

Our third case study is Fischer's protocol designed to ensure mutual exclusion among several processes competing for a critical section using timing constraints and a shared variable id. Fig. 9 shows the timed automaton for each process. The fault (depicted by dashed switch) can reset x_0, when process p_0 is in location req. A bad transition is one that reaches a state where more than one process is in the critical section:

$$bt = \{(s_0, s_1) \mid s_1 \models (\exists i, j.\ (i \neq j) \wedge (l_i = l_j = \text{critical}))\}$$

where l_i and l_j are the locations of processes i and j, respectively. By a simple observation, we can find out that if there is a process in waiting, and the fault happens, this may end up in violating the safety specification. The bounded response property we considered for this case study is

$$SPEC_{\overline{br}} = (\neg LS \mapsto_{\leq 3} Q) \ \wedge \ (Q \mapsto_{\leq 1} LS)$$

where $Q = (l_0 = \text{req}) \ \wedge \ (x_0 \geq 3)$. One possible set of legitimate states is the set of states reachable from the initial state (where all processes are in init and $id = 0$) by the switches of the timed automata.

Table 3 presents the results for 2 and 3 processes. Note that the number of zones increases so fast with the increase of processes in this case study, as we are dealing with both *distribution* and real-time nature of the system[7]. Although, we could run this case study for

7. We note that distribution (i.e., the ability of processes to read and write certain variables) adds an additional exponential blow-up to the synthesis problem [21]

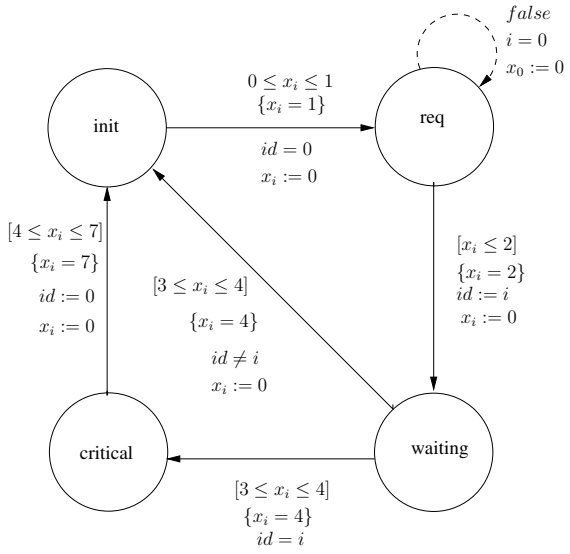

Fig. 9. Automaton for Fischer's mutual exclusion

TABLE 3
Results for Fischer's mutual exclusion

	2	**3**
Steps 1,2 (sec)	0.01	0.62
Step 3 (sec)	0.02	358.7
Step 4 (sec)	0.01	521.53
Step 5 (sec)	0.02	139.279
Step 6 (sec)	0.0	0.44
Total synthesis time (sec)	0.06	1020.569
Zone Graph Generation of Intolerant Model	0.0 s	0.0 s
Zone Graph Size of Enhanced Automaton	136	3978
Zone Graph Size of Intolerant Model	183	6122

small numbers of processes, we should note that we are dealing with the synthesis problem on a space with huge number of zones, and the fact that we could automate the process even for small numbers of processes is significant. Also, having access to a fault-tolerant system for a small number of processes can give key insights to designers to generalize the solution for any number of processes. For example, our method can be applied in cases where there exists a cut-off point [22].

Our conclusion is that in some case studies (such as our first one), the proposed algorithm competes with the verification time (model checking of the original model). In this case study, as the algorithm bottleneck is the zone graph generation time, we can claim that its scalability is as well as the zone graph generation in the underlying tool (IF in our case studies). In the case studies that our pruning strategy does not help (such as our second case study), the bottleneck of the algorithm is mostly in the recovery addition phase. Our idea for ranking the zones and updating the ranks dynamically has helped significantly to make this step more efficient.

7 Related Work

The objective of fault diagnosis in timed automata [23] 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 automaton.

Controller synthesis of timed systems is in spirit close to our work. Maler, et al. [7] 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.

In [24], a similar problem is tackled, with the difference that the controller has the option of doing nothing and let time pass, in addition to choosing among actions. We made a similar decision to let the program have any possible delay when it is not in a legitimate state.

An on-the-fly algorithm on synthesizing timed models using zone graphs is proposed in [25], which is implemented in the tool UPPAAL-TIGA [14]. The algorithm is a symbolic extension of the on-the-fly algorithm suggested by Liu and Smolka [26]. The main idea of this work is to (1) use a combination of a forward algorithm and backward propagation, which terminates as soon as a winning strategy is identified, and (2) 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 technique with these work (and also with [27]) is handling bounded response properties and, more importantly, adding recovery paths that the zone graph of the original model does not contain. Note that addition of recovery transitions is the key element in synthesizing a fault-tolerant timed automaton. Adding this step to the algorithm proposed in [25] will have similar challenges that we faced and solved in this work, as the underlying data structure in both is a zone graph. Moreover, adding this step to forward algorithm, and backward propagation as used in [25] may be more challenging. In addition, handling bounded response properties may be possible in UPPAAL-TIGA, if disabling clocks and urgent transitions are supported.

Game theory is another research line that is close to the automated synthesis of fault-recovery. In game-theoretic approaches, the synthesis of controllers and reactive programs are considered as a two-player game between the program and the environment. The interaction of the program and the environment is through a set of interface variables. Hence, the environment is only allowed to change the value of interface variables, while in the synthesis of fault-recovery, faults can change the value of any variable. The other difference is that in most of two-player game models, the set of states from where the first player (program) can make a move is disjoint

from the set of states from those that the second player can move from [28], while in our work, faults can occur in any state of the system. Similar to controller synthesis, the other distinction of our work with game theoretic approaches is that the latter does not address the issue of addition of recovery.

Automated addition of fault-tolerance to timed models has been studied with special focus on complexity analysis [9], [11]. The algorithms in [11] are proposed for the purpose of analyzing the time complexity of synthesis of different types of 2-phase recovery, and they are essentially impractical due to the use of region graphs as the finite representation of timed automata. 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/or logical specifications. Automated synthesis is known to be a notoriously difficult problem due to the high complexity of its 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 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 ensured with the fact that our algorithm adds no transition originating from a legitimate state of the input model. We used a space-efficient representation of timed models, known 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 (zone graph generation time for the input model using the IF toolset). Note that synthesis is a significantly more complex problem compared to model checking.

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 the 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.

9 Acknowledgment

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012.

References

[1] E. A. Emerson and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, 1982.

[2] A. Arora, P. C. Attie, and E. A. Emerson, "Synthesis of fault-tolerant concurrent programs," in *Principles of Distributed Computing (PODC)*, 1998, pp. 173–182.

[3] P. Cerný, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh, "Quantitative synthesis for concurrent programs," in *Computer Aided Verification (CAV)*, 2011, pp. 243–259.

[4] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Better quality in synthesis through quantitative objectives," in *Computer Aided Verification (CAV)*, 2009, pp. 140–156.

[5] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 404–415, 2006.

[6] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of IEEE, Special Issue on Discrete Event Dynamic Systems*, vol. 77, no. 1, pp. 81–98, 1989.

[7] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems," in *Proceedings of the Theoretical Aspects of Computer Science (STACS)*, 1995, pp. 229–242.

[8] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad, "Symbolic synthesis of masking fault-tolerant programs," *Springer Journal on Distributed Computing (DC)*, vol. 25, no. 1, pp. 83–108, 2012.

[9] 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)*, 2006, pp. 122–136.

[10] A. Girault and É. Rutten, "Automating the addition of fault tolerance with discrete controller synthesis," *Formal Methods in System Design (FMSD)*, vol. 35, no. 2, pp. 190–225, 2009.

[11] B. Bonakdarpour and S. S. Kulkarni, "Synthesizing bounded-time 2-phase fault recovery," *Springer Journal of Formal Aspects of Computing (FAOC)*, vol. 27, no. 1, pp. 1–31, 2015.

[12] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[13] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *International workshop on Automatic verification methods for finite state systems*, 1990, pp. 197–212.

[14] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Time for playing games!" in *Computer Aided Verification (CAV)*, 2007, pp. 121–125.

[15] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier, "IF: An intermediate representation and validation environment for timed asynchronous systems," in *World Congress on Formal Methods*, 1999, pp. 307–327.

[16] S. Bornot, J. Sifakis, and S. Tripakis, "Modeling urgency in timed systems," in *Compositionality: The Significant Difference (COMPOS)*, 1997, pp. 103–129.

[17] T. A. Henzinger, "Sooner is safer than later," *Information Processing Letters*, vol. 43, no. 3, pp. 135–141, 1992.

[18] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985.

[19] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, 2003, pp. 87–124.

[20] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *Concurrency Theory (CONCUR)*, 2005, pp. 66–80.

[21] B. Bonakdarpour and S. S. Kulkarni, "Revising distributed UNITY programs is NP-complete," in *International Conference on Principles of Distributed Systems (OPODIS)*, 2008, pp. 408–427.

[22] S. Jacobs and R. Bloem, "Parameterized synthesis," *Logical Methods in Computer Science*, vol. 10, no. 1, 2014.

[23] S. Tripakis, "Fault diagnosis for timed automata." in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2002, pp. 205–224.

[24] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis, "Controller synthesis for timed automata," in *IFAC Symposium on System Structure and Control*, 1998, pp. 469–474.

[25] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *Concurrency Theory (CONCUR)*, 2005, pp. 66–80.

[26] X. Liu and S. A. Smolka, "Simple linear-time algorithms for minimal fixed points (extended abstract)," in *International Colloquium on Automata, Languages and Programming (ICALP)*, 1998, pp. 53–66.
[27] O. Maler, D. Nickovic, and A. Pnueli, "On synthesizing controllers from bounded-response properties," in *Computer Aided Verification (CAV)*, 2007, pp. 95–107.
[28] N. Wallmeier, P. Hütten, and W. Thomas, "Symbolic synthesis of finite-state controllers for request-response specifications," in *Implementation and Application of Automata (CIAA)*, 2003, pp. 11–22.

Fathiyeh Faghih received her B.Sc. and M.Sc. degrees in Computer Engineering (Software) from Sharif University of Technology, Iran, in 2007 and 2009. In 2009, she joined the School of Computer Science at the University of Waterloo, Canada, where she received her Ph.D. in 2015. She is currently a postdoctoral fellow at McMaster University, Canada. Her research interests include formal methods, automated synthesis, fault-tolerance, and self-stabilizing systems.

Borzoo Bonakdarpour is currently an assistant professor of Computing and Software at McMaster University, Canada. He received his B. Sc. in Computer Engineering (Software) from the University of Esfahan, Iran, in 1999 and his M. Sc. and Ph.D. degrees in Computer Science from Michigan State University in 2004 and 2009. He has developed several efficient automated synthesis techniques that make synthesis of complex fault-tolerant distributed protocols possible. His Ph.D. dissertation on this subject was nominated for the ACM Doctoral Dissertation Award. His research interests include dependability and fault-tolerance, runtime monitoring, and program synthesis.