Supervisory Control Synthesis of Timed Automata Using Forcible Events

Aida Rashidinejad, Michel Reniers, Senior Member, IEEE, and Martin Fabian, Senior Member, IEEE

Abstract—This article presents an algorithm for synthesizing a supervisor for timed automata (TA) using the conventional supervisory control theory. The algorithm is directly applicable to TA without explicit transformation into finite automata, and iteratively strengthens the guards of edges labeled by controllable events and invariants of locations where the progression of time can be preempted by forcible events. The synthesized supervisor, also a TA, is controllable, maximally permissive, and guarantees a non-blocking and safe supervised plant. The use of real-valued clocks in TA makes it a practical modeling framework; however, the infinite state space brings challenges. The proposed algorithm addresses these by providing a synthesis method that avoids the state-space explosion of finite automata and the loss of information that can result from abstraction of real-time values.

Index Terms—Discrete-event systems (DES), forcible event, maximally permissive, nonblocking, real time, supervisory control, synthesis.

I. INTRODUCTION

Supervisory control theory [1] is an approach to synthesize supervisors for discrete-event systems (DES) [2] that restrict the behavior of a controlled plant to a desired behavior, the specification. The supervisor is typically guaranteed to be controllable, nonblocking, and maximally permissive [3]. An SCT is typically used with systems, such as communication networks and manufacturing systems, that can be modeled using finite automata (FA), or FA extended with variables, guards, and actions, so called extended finite automata (EFA) [4]. The dynamics of DES depend only on the ordering of the event occurrences, and so are independent of time [2], which makes FA unsuitable for real-time applications. To overcome this, different ways to introduce time to DES have been proposed, such as timed discrete-event systems (TDES) [5] and timed automata (TA) [6], which have dynamics dependent also on timing constraints on the events.

TDES incorporate discrete time in modeling DES by including a specific tick event that represents the passage of a unit of time. As the tick spontaneously occurs in the system it is considered uncontrollable and cannot be disabled by a supervisor. However, it is assumed that tick is preemptable by forcible events, and the execution of events (other than tick) is restricted within a specified lower and an upper time bound. Taking the nature of tick into account, SCT of DES, has been modified for TDES [5]. TDES has been further extended to timed interval DES [7]. Moreover, like DES, the model of TDES has been extended with discrete variables into timed EFA [8]. TA incorporate dense time by a finite set of real-valued clocks [9]. A clock constraint, called an invariant, is associated to each location and determines the maximal time that the system is allowed to stay in that location. Each edge between two locations is labeled by an event, a clock constraint called the guard, and a set of clocks that are reset to zero upon the occurrence of that event. Compared to TDES, TA bring a more natural modeling framework for real-life applications since: 1) it considers dense time, so it copes with the state-space explosion problem introduced by discrete time modeling, which is especially important for systems with variant time scales; and 2) it allows events to have multiple and different timing constraints, rather than specifying the time of each event occurrence by fixed lower and upper bounds.

Although TA and its accompanying analysis methods were originally introduced for model checking temporal logic properties [6], it has later been used to synthesize supervisors.

A game-based synthesis approach that works on the state space of a TA is presented by [10]. The synthesized supervisor decides for each configuration whether to execute an event or to allow time to pass. The supervisor is controllable and nonblocking, but it is not maximally permissive.

In general, the control of TA is challenging due to the clock variables, making the state space of TA infinite. To overcome this problem, some methods abstract TA into FA, and apply supervisory control synthesis on the abstraction [11, 12], synthesizing an untimed supervisor that is then refined by adding timing information to it.

In [13, 14], a transformation is introduced to obtain from a TA a minimal FA suitable for synthesis. The transformation is based on two special events, Set and Exp, where Set represents the set and reset of a clock, and Exp indicates the expiration of the clock. The SetExp-transformation [14] results in a minimal FA, for which a supervisor with forcible events is synthesized. However, it is currently unknown how to refine the synthesized supervisor (as an FA with Set and Exp events) to a TA (with these events translated into time constraints), and so the synthesis based on SetExp-transformation is not satisfying.

Supervisory control of TA using forcible events is also investigated in [15], where region-based abstraction is used to abstract a TA into an FA. For the FA, a synthesis algorithm is proposed. The synthesized supervisor is transformed back into a TA using a time-refinement technique. Although this method gives the supervisor as a TA, it still suffers from the state-space explosion problem caused by the abstraction.

In general, region-based abstraction results in a finite but huge FA [13, 16]. To overcome the state-space explosion problem of region-based abstraction, some state-space minimization methods have been proposed, such as zone-based abstraction [6, 17]. In [17], synthesis is performed on a timed game automaton, which is not a timed automaton but a zone-based representation of the state space underlying a TA. Also, the obtained supervisor is maximally permissive, but it is a timed game automaton, not a TA.
This article provides supervisor synthesis for TA such that:
1) an algorithm is proposed that works with automata instead of languages to ease integration of an implementation in a tool set, such as CIF or Supremica [18, 19];
2) the synthesized supervisor is maximally permissive, as well as controllable, and nonblocking;
3) the concept of forcible events from TDES is used to provide a more comprehensive result; and
4) the notion of clock regions of TA is adapted in a specific way for supervisory control synthesis.

The synthesis technique is similar to supervisor synthesis for EFA. The main differences being: 1) an EFA has a set of variables with finite domains, while a TA has real-valued clock variables, and 2) a TA includes location invariants that force the TA to leave the location before the invariant is violated. Dealing with real-valued clock variables and location invariants makes the synthesis of TA much more complex than the synthesis of EFA.

An earlier version of this work has been published as [20]. Compared to [20] this article generalizes the approach for control requirements that are given as automata, and applies the method to a well-known case study. Technical lemmas and proofs are given in an extended version of this article [21].

II. PRELIMINARIES

A TA is an FA extended with a finite set of real-valued clocks. To model the timing behavior of TA, the accepting temporal conditions to switch between different modes (locations) or stay in the current one are represented by clock constraints [6, 22].

Definition 1 (Clock constraints [22]): Given a finite set of real-valued clocks \( C \), \( x \sim n \) and \( x - y \sim n \) are atomic clock constraints for any \( x, y \in C, \sim \in \{<, =, >\} \), and \( n \in \mathbb{N} \). Clock constraints are defined as follows. Any atomic clock constraint is a clock constraint, and for any two clock constraints \( \phi_1 \) and \( \phi_2 \), also \( \phi_1 \land \phi_2 \) and \( \phi_1 \lor \phi_2 \) are clock constraints.

Instead of writing \( x - x = 0 \) with \( x \in C \) as a clock constraint, we write \( true \). Similarly, \( false \) is written instead of \( x - x > 0 \).

Definition 2 (Clock valuation): Given a set of clocks \( C \), a clock valuation \( u : C \rightarrow \mathbb{R}_{\geq 0} \) assigns a real value to each clock \( x \in C \).

A clock valuation \( u \) satisfies a clock constraint \( \phi \), denoted \( u \models \phi \), if \( \phi \) is \( true \) for the values assigned by \( u \) to each clock.

Definition 3 (Deterministic timed automaton [6]): A deterministic timed automaton is a 7-tuple \((C, L, \Sigma, E, L_0, l_0, I)\) where:
1) \( C \) is a finite set of clocks with a nonnegative real value (from \( \mathbb{R}_{\geq 0} \)).
2) The initial value of each clock variable is always assumed to be 0.
3) \( L \) is a finite set of locations;
4) \( \Sigma \) is a finite set of events;
5) \( E \) is a finite set of edges with elements of the form \((l_s, \sigma, g, r, l_t)\) for which \( l_s, l_t \in L \) are the source and target locations, respectively, \( \sigma \in \Sigma \), clock constraint \( g \) is the guard, and \( r \subseteq C \) is the set of clocks to be reset to 0. For any pair of different edges \( e_1, e_2 \in E \), with the same source location \((e_1, l_s = e_2, l_t)\) and labeled by the same event \((e_1, \sigma = e_2, \sigma)\), the clock constraints are mutually exclusive \((e_1, g \land e_2, g = false)\).
6) \( L_m \subseteq L \) is the set of marked locations;
7) \( l_0 \in L \) is the initial location;
8) \( I \) is a function associating an invariant to each location \( l \in L \). An invariant is a clock constraint that has to be satisfied when the system is in the location.

In [22], guards are generally given as clock constraints, but invariants are restricted to downwards closed clock constraints: \( x < n \) or \( x \leq n \). In this work, similar to [23], both guards and invariants are allowed to be arbitrary clock constraints.

For TA, we frequently use the following notations.
1) The notation, is used to refer to an element of a tuple. For instance, \( e, \sigma \) refers to \( \sigma \) from the edge \( e \in E \).
2) The notation \( pred^g \), for a predicate \( pred \) and an increase \( \delta \in \mathbb{R}_{\geq 0} \), denotes the predicate where all occurrences of the variables \( x \in C \) are replaced by \( x + \delta \). For instance, \((x \geq 3)^6 \) gives \( x + \delta \geq 3 \).
3) The notation \( pred[r] \), for a predicate \( pred \) and a reset \( r \), denotes the predicate in which all occurrences of clock variables from \( r \) are replaced by zero.
4) The notation \( Preds(C) \), for the set of all predicates over the clock variables from \( C \). This set contains all atomic clock constraints and logical combinations of these.
5) The notation \( P_{\Sigma}(w) \), for \( \Sigma \subseteq \Sigma \) and \( w \in \Sigma^* \), indicates the natural projection of \( w \) to a word from \( \Sigma^* \), as defined in [2]. The natural projection is extended for a given language \( L \subseteq \Sigma^* \) as \( P_{\Sigma}(L) := \{ P_{\Sigma}(w) \in \Sigma^* \mid w \in L \} \).

In the examples, TA are depicted graphically. The locations are represented by circles and the edges by arrows from the source location to the target location, labeled with the event, the guard and the reset. The reset of a clock \( x \in r \) is denoted by \( x := 0 \). Invariants of locations are indicated inside the locations. Absence of an invariant in a location represents the invariant that always holds. The initial location is depicted by a dangling incoming arrow, and the marked locations by double circles.

To illustrate each step of the approach, the bus–pedestrian example from [5] is used throughout this article.

Example 1 (Bus–pedestrian): Imagine that a bus is headed directly for a pedestrian and will run over him at time \( x = 2 \) if he does not move. The pedestrian needs an amount of time \( y = 1 \) to realize his fate, after which he has the chance to jump out of the bus’s path. If the pedestrian jumps before the bus passes, he is safe. Fig. 1 gives the automata, representing the bus, the pedestrian, and the safe behavior of the system. The safe behavior is modeled in such a way that if the pedestrian jumps before the bus passes, then the system goes to a marked state. Otherwise, the system goes to a blocking state.

Definition 4 (Subautomaton of a TA): Given a TA \( A = (C, L, \Sigma, E, L_0, l_0, I) \), a TA \( B = (C, L', \Sigma, E', L'_m, l'_0, I') \) is a sub-automaton of \( A \), denoted \( B \subseteq A \), if:
1) \( L' \subseteq L \);
2) for all \( (l_s, \sigma, g', r, l_t) \in E' \) : \((l_s, \sigma, g, r, l_t) \in E \) for some \( g \) such that \( g' \Rightarrow g \);
3) \( L'_m = L_m \cap L' \);
4) for all \( l \in L' \) \( I'(l) \Rightarrow I(l) \).

Applications are typically modeled by a network of automata, where each automaton represents a single component or subsystem; compare Fig. 1. A single automaton representing the network of automata can then be generated as the synchronous product of the constituent automata.

In [6] and [22], synchronous product of TA is defined under the assumption that the two TA do not share clock variables. This is is
relaxed here, and the synchronous product is generalized to TA that may share clock variables. This is inspired from the synchronous product of EFA as defined in [4]. If two TA share a clock variable, then this clock variable can only be reset if both TA do so. A consequence of this is that such a reset can only occur on an edge with a shared event.

Definition 5 (Synchronous product of TA): The synchronous product of two TA $G_1 = (C_1, L_1, \Sigma_1, E_1, l_{10}, l_1)$ and $G_2 = (C_2, L_2, \Sigma_2, E_2, l_{20}, l_2)$, is given by $G_1 \times G_2 = (C_1 \cup C_2, L_1 \times L_2, \Sigma_1 \times \Sigma_2, E_1 \times E_2, l_{10} \times l_{20}, l_1 \times l_2)$. where for each $l_1 \in L_1$ and $l_2 \in L_2$, $I_{l_1} = (l_1, l_2)$ and each edge in $E_1 \times E_2$ is as follows.

1) $\sigma \in \Sigma_1 \times \Sigma_2$, then for every $(l_{11}, \sigma, g_1, r_1, t_{11}) \in E_1$ with $r_1 \cap C_1 = \emptyset$ and $l_2 \in L_2$, $(l_{12}, \sigma, g_1, t_{12}) \in I_{l_2}$.
2) $\sigma \in \Sigma_1 \times \Sigma_2$, then for every $(l_{12}, \sigma, g_1, r_2, l_{12}) \in E_2$ with $r_2 \cap C_1 = \emptyset$ and $l_1 \in L_1$, $(l_{11}, \sigma, g_2, t_{11}) \in I_{l_1}$.
3) $\sigma \in \Sigma_1 \times \Sigma_2$, then for every $(l_{11}, \sigma, g_1, r_1, l_{11}) \in E_1$ and $(l_{12}, \sigma, g_2, r_2, l_{12}) \in E_2$ with $r_1 \cap C_2 = r_2 \in C_1$, $(l_{13}, l_{12}) \in I_{l_2}$.

For the bus–pedestrian example, the synchronous product of the three automata is shown in Fig. 2.

Every TA has an underlying semantic graph [6], [16].

Definition 6 (Semantic graph): The semantic graph of a TA $G = (C, L, \Sigma, E, l_0, l, I)$, is a labeled graph with a set of states $X \subseteq C \times (C \rightarrow \mathbb{R}_{\geq 0})$, consisting of a location $l$ and a clock valuation $u$ such that $(l, u) \in X$ iff $u \models I(l)$. The initial state is $(l_0, 0)$ if $0 \models I(l_0)$, otherwise, the semantic graph is undefined. The semantic graph has the following transitions.

1) Event transition: from state $(l_1, u_1)$ to state $(l_1, u_1[x])$ labeled by event $\sigma$ if there is an edge $e = (l_1, \sigma, g, r, l_1)$ such that $u_1 \models g$ and $u_1[x] \models I(l_1)$.
2) Time transition: from state $(l, u)$ to state $(l, u + \Delta)$ labeled with delay $\Delta \in \mathbb{R}_{\geq 0}$ if $u + \Delta \models I(l)$ for any $\Delta$ such that $0 \leq \Delta \leq \Delta$. Note that for a valuation $u$ and a real value $\delta$, $u + \delta$ denotes the clock valuation with $u + \delta(x) = u(x) + \delta$ for each clock $x \in C$.

Moreover, states $(l, u)$ in the semantic graph with $l \in L_{\text{max}}$ (regardless of the clock valuation $u$) are marked. A state $(l, u)$ in the semantic graph of $G$ is said to be reachable if there exists a sequence of transitions from the initial state to $(l, u)$. A word $w$ is a finite sequence of labels; $w \in (\Sigma \cup \mathbb{R}_{\geq 0}^*)$ with $\epsilon$ denoting the empty sequence. The language of $G$, denoted $L(G)$, is the set of all words obtained by concatenating the labels of subsequent transitions in its semantic graph starting from the initial state. The marked language of $G$, denoted $L_{\text{max}}(G)$, is the set of all words from $L(G)$ ending in a marked state. Note that for any $G' \subseteq G$, $L(G') \subseteq L(G)$.

Note that since a TA is allowed to have arbitrary clock constraints as invariants, it may be the case that $0 \not\models I(l_0)$. This may happen due to modeling issues, or through synthesis in which case no supervisor exists.

Based on the semantic graph, some relevant notions for TA are defined.

Definition 7 (Nonblockingness): A state in a semantic graph is nonblocking if there exists a path from that state to a marked state, i.e., a state $(l_{1n}, u_{1n})$ with $l_{1n} \in L_{\text{max}}$. A TA is nonblocking if all reachable states in its semantic graph are nonblocking.

In the following, the plant is given as a TA $G$ represented by $(C, L, \Sigma_G, E_G, l_0, l_G)$. It is assumed that all events are observable. However, not all of the events might be controllable. The set of events $\Sigma_G$ is partitioned into a set of uncontrollable events $\Sigma_{uc}$ and a set of controllable events $\Sigma_c = \Sigma_G \setminus \Sigma_{uc}$. Uncontrollable events are events that occur spontaneously in the plant such as disturbances or sensor readings. Controllable events are signals sent to the actuators. Time passage is uncontrollable by nature. However, it may be pre-empted by execution of a forcible event $\sigma_j \in \Sigma_{for}$, where $\Sigma_{for} \subseteq \Sigma_G$. Consequently, considering the semantic graph of a TA, a time transition enabled at a state is considered uncontrollable by default, unless there is also a forcible event transition enabled at that state. Then, the time transition is said to be preemptable. Note that forcing is an issue that is related to the modeling of the plant. A supervisor only decides whether forcing events is appropriate or not, a forcible event can be controllable or uncontrollable, as discussed in [3]. An example of an uncontrollable forcible event is the landing of a plane where air defense could force the plane to land within some time but not prevent it from landing eventually [5]. In figures of TA, edges labeled by uncontrollable events are indicated by dashed lines, edges labeled by controllable events are indicated by solid lines, and forcible events are underlined. For the bus–pedestrian example, the event pass is uncontrollable, and the event jump is controllable and forcible.

The following definition of controllability for TA with forcible events, is inspired from [5].

Definition 8 (Controllability of TA with forcible events): Given a plant $G$ with uncontrollable events $\Sigma_{uc}$, and forcible events $\Sigma_{for}$, a TA $S$ is controllable w.r.t. $G$ if for all $w \in L(S)\mid G$ and $\sigma \in \Sigma_{uc} \cup \mathbb{R}_{\geq 0}$, whenever $w\sigma \models L(G)$:

1) $w\sigma \models L(S)\mid G$ or
2) $\sigma \in \mathbb{R}_{\geq 0}$ and $w\sigma^\delta \models L(S)\mid G$ for some $\sigma^\delta \in \mathbb{R}_{\geq 0}$ with $\sigma' < \sigma$ and $\sigma \in \Sigma_{for}$.

Property (1) mentioned previously is the standard controllability property; $S$ cannot disable uncontrollable events that $G$ may generate. However, if a forcible event is enabled, this may preempt the time event, which is captured by Property (2).

A supervisor $S$ is called proper for a plant $G$, if $S$ is controllable w.r.t. $G$, and the supervised plant $S\mid G$ is nonblocking.

Definition 9 (Maximally permissiveness): A proper supervisor $S$ is maximally permissive for a plant $G$, if $S$ preserves the largest admissible behavior of $G$ compared to any other proper supervisor $S'$; for any proper $S': L(S')\mid G \subseteq L(S)\mid G$.

As stated in [6], the clock valuations of a TA $G$ can be divided into a finite set of clock regions using the definition of region equivalence. Here, we introduce extended clock regions of a TA $G$, denoted $R_G$.

Definition 10 (Extended clock regions of TA): Consider TA $G$ with a set of clocks $C$. The clock ceiling function $k: C \rightarrow \mathbb{N}$ of this TA gives, for each clock $x \in C$, the largest natural number that $x$ is compared to in atomic constraints of the form $x \sim n$ and $x - y \sim n$ in guards or invariants of that TA.

Each clock region $r_{C} \in R_G$ is specified by the following.

1) For each clock $x \in C$, a single clock constraint of one of the following forms:
   a) $x = n$ for some $n \in \{0, \ldots, k(x)\}$;
   b) $n - 1 < x < n$ for some $n \in \{1, 2, \ldots, k(x)\}$; or
   c) $x > k(x)$.

2) For any two different clocks $x, y \in C$, a single clock constraint of one of the following forms:
   a) $y - x + k(x) = q$ for some $q \in \{0, \ldots, k(x) + k(y)\}$;
   b) $q - 1 < y - x + k(x) < q$ for some $q \in \{1, \ldots, k(x) + k(y)\}$;
   c) $y - x + k(x) < 0$;
   d) $y - x + k(x) > k(x) + k(y)$.
Extended clock regions from Example 2.

Note that $k(x)$ does not restrict the value of the clock variable $x$; it only gives the largest number that $x$ is bounded to by guards or invariants. Considering Fig. 2, $k(x) = 2$. However, in location $(g, r, \perp)$, the value of $x$ can grow to any real number larger than or equal to 2. We call a clock region unbounded if it is related to $x > k(x)$ for some $x \in C$. Otherwise, the region is called bounded.

Example 2: Fig. 3 depicts the extended clock regions for a TA with two clock variables $x, y$, where $k(x) = 2$ and $k(y) = 1$. The construction of the extended clock regions is similar to the region constructions used in [6] and, for updatable TA, in [24]. The clock regions as used in [6] are indicated in black. Our partitioning into extended regions (Definition 6) is finer than the ones from [6] in the sense that some of their unbounded regions are replaced by multiple extended regions. For instance, in Fig. 3 the region $0 < x < 1, y > 1$ exists in the approach of [6], but is replaced by the three extended regions $0 < x < 1, y > 1, y < x < 1$; $0 < x < 1, y > 1, y - x = 1$; and $0 < x < 1, y > 1, y - x > 1$. As a consequence any extended clock constraint can be related to a subset of regions, which is not the case with the regions of [6]. The approach of [24] defines regions that are compatible with more complicated updates than only resets. Another difference is that in [24] regions are defined based on the updates present in the automaton, whereas we define them only based on the upper bound values of the clocks in the automaton.

Note that although the number of the extended clock regions is larger than the number of clock regions, it is still finite because the set of clock regions is finite (see [6] for details), and the extended clock regions include all the unbounded regions from the set of clock regions, and it partitions each unbounded region into a finite number of new regions.

III. BASIC TSC SYNTHESIS

A. Problem Formulation

The basic TSC synthesis problem is: Given a plant model $G$ as a TA, synthesize a timed supervisor $S$ as a TA, such that:

1) $S \parallel G$ is controllable w.r.t. $G$ (Definition 8);
2) $S \parallel G$ is nonblocking (Definition 7); and
3) $S$ is maximally permissive, (Definition 9).

Consider the bus–pedestrian example, a supervisor is required to avoid reaching the blocking location $(g, r, \perp)$ in Fig. 2. To synthesize such a supervisor, it is needed to determine the bad states $(l, u)$ in the semantic graph that should be made unreachable. These are: 1) states that are blocking, and 2) states that reach a bad state through an uncontrollable event or a time transition that cannot be preempted. We need to determine the clock valuations for which a location of a TA is a bad state. For this purpose, we start by determining the clock valuations for which a location is nonblocking, referred to as the “nonblocking predicate” of a location. Based on the nonblocking predicate, a “bad state predicate” is associated to each location determining the clock valuations for which the location is mapped to a bad state in the semantic graph.

Algorithm 1: Nonblocking Predicate (NBP).

1: $i := 0$
2: for $l \in L$ do $N^0(l) := I_G(l)$, if $l \in L_m$, $\text{false}$, otherwise.
3: repeat
4: for $l \in L$ do
5: $N^{i+1}(l) := N^i(l) \lor \forall_{r', r_1, r_2} (g \land I_G(l)[r] \land N^i(l)[r_1]) \lor$
6: $\exists \Delta N^i(l)^\Delta \land \forall \delta \leq \Delta : I_G(l)^\delta$
7: until $i = i + 1$
8: $l \in N^i(l) := N^{i-1}(l)$
9: for $l \in L$ do $N(l) := N(l)$

B. Nonblocking Condition

Given a plant $G$, Algorithm 1 associates a nonblocking predicate $N(l)$ to each location $l \in L$. Initially (line 2), $N^0(l)$ with $i = 0$ is set to $I_G(l)$ if $l$ is a marked location, and to false otherwise. The nonblocking predicate of each location is updated (line 4) to $N^{i+1}(l)$ based on:

1) the current nonblocking predicate $N^i(l)$;
2) the condition for any outgoing edge $(l, \sigma, g, r, l')$ to lead to a nonblocking location (an event transition leading to a nonblocking state in the semantic graph); and
3) the condition to stay (for some time delay $\delta \leq \Delta$) in a nonblocking location as long as the invariant is satisfied (represented by a time transition leading to a nonblocking state in the semantic graph). This iterates until a fix point is reached where the nonblocking predicate stays the same for all locations (line 6).

Algorithm 1 follows the same steps as presented for the nonblocking predicate of EFA in [25] with the following adjustments (indicated in red in Algorithm 1).

1) The initial nonblocking condition for marked locations is set to the location invariant $I_G(l)$ instead of true. This is to take into account the invariants of the marked locations.
2) In the update (line 4), the invariant of the target location is added to the second term to guarantee that the invariant of the target location is satisfied upon entering that location.
3) The third term is added to take into account the time transitions in the semantic graph of the TA that may be used for reaching a nonblocking state.

Algorithm 1 does not only say whether a marked location can be reached using backward reachability computation methods as defined for instance in [26], but it also determines the condition under which a marked location can be reached. These conditions are subsequently needed to determine the bad state predicates, which in turn may ultimately be needed in adapting the guards of edges and the invariants of locations.

Property 1 (NBP termination): Algorithm 1 terminates.

Property 2 (NBP and nonblocking states): Given a plant $G$ and NBP$(G)$; for any $(l, u)$ in (the semantic graph of) $G$, $(l, u)$ is a nonblocking state if $u \models N(l)$, where $N = \text{NBP}(G)$.

Example 3 (Nonblocking predicate for bus–pedestrian): Consider the bus–pedestrian from Example 1. The result of Algorithm 1 is given in Table 1. The conditions for locations $(g, r, \perp)$ and $(g, c, 2)$...
are left out, as they are false and true, respectively, for all iterations. Note that for presentation purposes we sometimes simplify a predicate as is done in Table (x ≤ 2 ∧ (y ≥ 1 ∨ x − y ≤ 1) is simplified to x ≤ 2 ∧ x − y ≤ 1), since they represent the same clock valuations.

C. Bad State Condition

Given a plant G, and the nonblocking predicate computed by Algorithm 1, Algorithm 2 associates a bad state predicate B(l) to each location l ∈ L.

Initially, B(l) with i = 0 is set to the logical negation of N(l) for each location l ∈ L (line 2) because these characterize the blocking states. Then, the bad state predicate of each location is updated to B(l+1) (line 4) based on:

4) the previous bad state predicate B(l);
5) the condition of any outgoing edge (l, σ, g, r, l′) labeled by an uncontrollable event σ ∈ Σuc to lead to a bad state (an uncontrollable event transition leading to a bad state in the semantic graph); and
6) the condition of staying in a bad state for some time delay δ ≤ Δ as long as the invariant is satisfied for all the clock variables and while there is no forcible event able to preempt time for any δ′ ≤ δ (an uncontrollable time transition leading to a bad state in the semantic graph).

This iterates until a fix point is reached where the bad state predicate stays the same for all locations (line 6).

The differences (indicated in red) between Algorithm 2 and the bad state condition of EFA presented by [25] are as follows. 1) The invariant of the target location is determined if the uncontrollable transition should exist in the semantic graph. 2) The third term takes into account the nonpreemptible time transitions leading to a bad state.

Property 3 (BSP termination): Algorithm 2 terminates.

Property 4 (BSP and bad states): Given a plant G and NBP(G): for any (l, u) in the semantic graph of G, (l, u) is a bad state if u ⊨ B(l), where B = BSP(G, NBP(G)).

Example 4 (Bad state predicate for bus–pedestrian): By applying Algorithm 2 on the bus–pedestrian example, the bad state predicate of locations (a, r, 0) and (a, c, 1) are obtained as in Table II. The bad state predicates for (g, r, ⊥) and (g, c, 2) are true and false, respectively.

D. Synthesis

The synthesis procedure has two loops (see Algorithm 3):

1) Guard Adaptation: In Loop-1 the guards are adapted to obtain a supervisor that prevents the bad states; the guard of each edge (l, σ, g, r, l′) labeled by a controllable event σ ∈ Σc is adjusted to become (l, σ, g ∧ ¬B(l′))[r, r, l′].

2) Invariant Adaptation: So far, forcible events have not been taken into account. The effect of forcible events preemption time events is taken into account in the invariant adaptation (Loop-2). The invariant of a location l ∈ L can be changed only if there exists an edge labeled by a forcible event σf ∈ Σfor starting from l. In this case, the invariant is adapted to prevent reaching the bad states as follows:

I(l) := I(l) ∧ ¬B(l).

Algorithm 3 is the synthesis algorithm. For a TA G with a set of uncontrollable events Σuc, and a set of forcible events Σfor, it results in S = (C, L, ΣG, EΣ, Lm, l0, IS). The notation FΣ(l) = {e ∈ ES | e.lu = l, e.σ ∈ Σfor, e.gis satisfiable} gives the set of edges of S starting from location l and labeled by a forcible event. The algorithm starts with S = G. In the inner loop (lines 7–13), the guards of edges labeled by controllable events are updated until a fix point is reached. In the outer loop (lines 5–19), the invariants of locations where there exists an edge labeled by a forcible event are adapted until a fix point is reached. Otherwise, the synthesis goes back to Loop-1 (guard adaptation). Note that if the invariant of a location l is adapted, and in some later iteration the guard of an edge labeled by the forcible event becomes false, then the invariant should be set back to its original IΣc(l). This is captured in line 17.

Given a plant G, in case that u0 ⊨ B(l0), with B as the result of Algorithm 2 for TSCS(G) and NBP(TSCS(G)), then TSCS(G) is undefined. In the rest of this article, it is assumed that u0 ̸⊨ B(l0) for any given plant G.

### Table I

| N | Loc (a,r,0) | Loc (a,c,1) |
|---|-------------|-------------|
| 0 | false       | false       |
| 1 | false       | x = 2       |
| 2 | x = 2 ∧ y > 1 | x < 2       |
| 3 | x < 2 ∧ (y ≥ 1 ∨ x − y ≤ 1) | x < 2       |
| 4 | x ≤ 2 ∧ x − y ≤ 1 | x ≤ 2       |

### Table II

| B | j |
|---|---|
| 0 | x > 2 ∨ x − y > 1 |
| 1 | x > 2 ∨ x − y > 1 |
| 2 | x ≥ 2 ∧ x − y > 1 |

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
Algorithm 3: Timed Supervisory Control Synthesis (TSCS).

Input: \( G = (C, L, \Sigma_G, E_G, L_m, l_0, I_G), \Sigma_{ac}, \Sigma_G, \Sigma_{for} \).

Output: \( S = (C, L, \Sigma_G, E_S, L_m, l_0, I_S) \).

1: \( S := G \);
2: \( n := 0 \);
3: for \( e \in E_S, e = (l, \sigma, g, r, l') \) do \( e.g^0 := e.g \);
4: for \( l \in L \) do \( I_G(l) := I_G(l) \);
5: repeat \( \triangleright \) Loop-2: Invariant Adaptation
6: \( m := 0 \);
7: repeat \( \triangleright \) Loop-1: Guard Adaptation
8: \( N^n.m := NBP(S) \);
9: \( B^n.m := BSP(S, N^n.m) \);
10: for \( e \in E_S \) such that \( e.\sigma \in \Sigma_c \)
11: \( e.g^{m+1} := e.g^m \land \neg B^{n.m}(l')[r] \);
12: \( m := m + 1 \);
13: until \( \forall e \in E_S e.g^m = e.g^{m-1} \);
14: for \( e \in E_S e.g := e.g^m \);
15: for \( l \in L \) do
16: if \( F_S(l) \neq \emptyset \) then \( I^2_S(l) := I^2_S(l) \land \neg B^{n.m}(l) \);
17: else \( I^2_S(l) := I_G(l) \);
18: \( m := n + 1 \);
19: until \( \forall l \in L I^2_S(l) = I^2_S(l) \);
20: for \( l \in L \) do \( I_S(l) := I^2_S(l) \).

Property 5 (TSCS termination): Given plant \( G \); Algorithm 3 terminates, and \( S \) is a subautomaton of \( G \).

Theorem 1 (Controllability, nonblockingness, and maximal permissiveness): Given plant \( G \) and supervisor \( S = TSCS(G) \). \( S \) is controllable w.r.t. \( G \), \( S||G \) is nonblocking, and \( S \) is maximally permissive for \( G \).

Example 5 (Supervisor synthesis for bus–pedestrian): Let us apply Algorithm 3 to the bus–pedestrian from Example 1. Initially, \( S \) is set to the plant depicted in Fig. 2. First, the guard of the edge labeled by the controllable event \( jump \) is modified to \( y > 1 \land x < 2 \). Next, for \( I_0 := (a, r, 0) \), the invariant is adapted to \( x < 2 \land x < 2 \land x < y \leq 1 \). The synthesized supervisor is shown in Fig. 4.

Remark 1: Invariant adaptation can highly affect the synthesis result. Consider Example 5. Algorithm 3 does not give a supervisor without invariant adaptation.

IV. REQUIREMENT AUTOMATA

To generalize the method, we solve the TSC synthesis problem for a given set of control requirements. It is assumed that an allowed behavior of \( G \) is denoted by the timed automaton \( R = (C_R, Q, \Sigma_R, E_R, Q_m, q_0, I_R) \), where \( \Sigma_R \subseteq \Sigma_G \) and \( C_R \cap C = \emptyset \). We call a supervisor \textit{safe} w.r.t. \( G \) and a requirement if the supervised plant satisfies the control requirement.

Definition 11 (Safety): Given a plant \( G \) and a control requirement \( R \), a TA \( S \) is a safe w.r.t. \( G \) and \( R \) if \( P_{S,G}(L_m(S)[G]) \subseteq P_{S,R}(L_m(R)) \).

Requirement automata are considered in synthesis by synchronizing them with the plant. However, if a requirement automaton is not controllable (Definition 8), then it is necessary to let the supervisor know about the uncontrollable events that are disabled by a given requirement.

V. CASE STUDY

In this section, we consider the verification example from [6], [23] and modify it for synthesis. The TA representing the train and gate are taken from [6], [23] and are depicted in Fig. 5. The events \textit{app} and \textit{out} for the train, and the events \textit{down} and \textit{up} for the gate are assumed to be uncontrollable. Moreover, the events \textit{raise} and \textit{lower} of the gate are assumed to be forcible. The system in [6] and [23] also involves an automatic controller, to open and close the gate in a railroad crossing. The control requirements for the train–gate-controller system are as follows [6].

1) Safety: if the train is inside the gate, the gate is closed.
2) Liveness: the gate is never closed more than 10 time units.

In [6], [23], the system is assessed to be safe by analyzing the timing constraints: they say that with the (random) gate controller, that is part of the system, the event \textit{lower} always precedes the event \textit{in}, so the system is always safe. We do not consider such a controller to already be given as a part of the system. We synthesize a supervisor that is correct-by-construction, and more importantly this supervisor guarantees controllability, nonblockingness, and maximal permissiveness.

The safety requirement is represented by the left TA in Fig. 6, where the blue location and edges are added to make the TA complete. The liveness requirement is represented by the right TA in Fig. 6.

The supervisor synthesized by Algorithm 3 for the train–gate and control requirements is given in Fig. 7. In this figure, the synchronous product of the train–gate and control requirements is indicated in black and the adaptations made by the supervisor in red. Edges with guards equal to \textit{false} and locations reached by them have been removed. The supervisor obtained through synthesis allows more behavior than the (manually constructed) controller given in [6], [23]. For instance,
in their controller the event lower is not enabled before the train approaches, while this is allowed by the synthesized supervisor in Fig. 7.

VI. CONCLUSION AND FUTURE WORK

This article proposes a synthesis algorithm for TA with forcible events that can handle the passage of time. The algorithm works directly on TA without explicitly representing them as finite state automata. The algorithm strengthens guards of edges labeled by controllable events, and invariants of locations from which there exist edges labeled by forcible events. Based on extended clock regions, the synthesized supervisor is shown to be nonblocking, controllable, and maximally permissive. The problem is also generalized for a set of safety requirements modeled as TA, and it is shown that the synthesized supervisor satisfies controllability, nonblockingness, maximal permissiveness, and safety. Finally, the results are verified on a case study. Future work will investigate networked supervisory control of TA, and implementation of the proposed approach in existing tools.

REFERENCES

[1] P. J. Ramadge and W. M. Wonham, “The control of discrete event systems,” Proc. IEEE, vol. 77, no. 1, pp. 81–98, Jan. 1989.
[2] C. G. Cassandras and S. Lafortune, Introduction to Discrete Event Systems. Berlin, Germany: Springer, 2009.
[3] W. M. Wonham, “Supervisory control of discrete-event systems,” Encyclopedia of Syst. and Control, pp. 1396–1404, 2015.
[4] M. Skogstam, K. Åkesson, and M. Fabian, “Modeling of discrete event systems using finite automata with variables,” in Proc. IEEE 46th Conf. Decis. Control, 2007, pp. 3387–3392.
[5] B. A. Brandin and W. M. Wonham, “Supervisory control of timed discrete-event systems,” IEEE Trans. Automat. Control, vol. 39, no. 2, pp. 329–342, Feb. 1994.
[6] R. Alur and D. L. Dill, “A theory of timed automata,” Theor. Comput. Sci., vol. 126, no. 2, pp. 183–235, 1994.
[7] B. Brandin, R. Su, and L. Lin, “Supervisory control of time-interval discrete event systems,” IFAC-PapersOnLine, vol. 53, no. 4, pp. 217–222, 2020.
[8] S. Miremadi, Z. Fei, K. Åkesson, and B. Lennartson, “Symbolic supervisory control of timed discrete event systems,” IEEE Trans. Control Syst. Technol., vol. 23, no. 2, pp. 584–597, Mar. 2015.
[9] A. Dubey, “A discussion on supervisory control theory in real-time discrete event systems,” ISIS, vol. 9, 2009, Art. no. 112.
[10] O. Maler, A. Pnueli, and J. Sifakis, “On the synthesis of discrete controllers for timed systems,” in Ann. Symp. Theor. Aspects Comput. Sci., 1995, pp. 229–242.
[11] H. Wong-Toi and G. Hoffmann, “The control of dense real-time discrete event systems,” in Proc. IEEE 30th Conf. Decis. Control, 1991, pp. 1527–1528.
[12] S. Tripakis and K. Altisen, “On-the-fly controller synthesis for discrete and dense-time systems,” in Int. Symp. Formal Methods, 1999, pp. 233–252.
[13] A. Khoumsi and M. Nourelfath, “An efficient method for the supervisory control of dense real-time discrete event systems,” in Proc. 8th Int. Conf. Real-Time Comput. Syst., 2002.
[14] L. Ouedraogo, A. Khoumsi, and M. Nourelfath, “SetExp: A method of transformation of timed automata into finite state automata,” Real-Time Syst., vol. 46, no. 2, pp. 189–250, 2010.
[15] A. Rashidinejad, P. van der Graaf, and M. Reniers, “Nonblocking supervisory control synthesis of timed automata using abstractions and forcible events,” in Proc. 16th Int. Conf. Control, Automat., Robot. and Vis., 2020, pp. 1–8. doi: 10.1109/ICARCV50220.2020.9305312.
[16] S. Tripakis and S. Yovine, “Analysis of timed systems using time-abstraction bisimulations,” Formal Methods Syst. Des., vol. 18, no. 1, pp. 25–68, 2001.
[17] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis, “Controller synthesis for timed automata,” IFAC Proc. Vol., vol. 31, no. 18, pp. 447–452, 1998.
[18] D. A. van Beck et al., “CIF 3: Model-based engineering of supervisory controllers,” in Int. Conf. Tools and Algorithms for the Construction and Anal. of Syst., 2014, pp. 575–580, doi: 10.1007/978-3-642-54862-8_48.
[19] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, “Suprema—An integrated environment for verification, synthesis and simulation of discrete event systems,” in Proc. 8th Int. Workshop Discrete Event Syst., 2006, pp. 384–385.
[20] A. Rashidinejad, P. van der Graaf, M. Reniers, and M. Fabian, “Nonblocking supervisory control of timed automata using abstractions and forcible events,” in Proc. 15th Int. Workshop on Discrete Event Syst., 2020, pp. 1–8. doi: 10.1016/j.ifacol.2021.04.035.
[21] A. Rashidinejad, M. Reniers, and M. Fabian, “Supervisory control synthesis of timed automata using forcible events,” 2021, arXiv:2102.09338.
[22] J. Bengtsson and W. Yi, Timed Automata: Semantics, Algorithms and Tools. Berlin Germany: Springer, 2004, pp. 87–124.
[23] R. Alur, “Timed automata,” in Int. Conf. Comput. Aided Verification, 1999, pp. 8–22.
[24] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit, “Updatable timed automata,” Theor. Comput. Sci., vol. 321, no. 2–3, pp. 291–345, 2004.
[25] L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson, “Nonblocking and safe control of discrete-event systems modeled as extended finite automata,” IEEE Trans. Automat. Sci. Eng., vol. 8, no. 3, pp. 560–569, Jul. 2011.
[26] P. Bouyer and F. Laroussinie, “Model checking timed automata,” Model Verification Real-Time Systems: Formalisms Softw. Tools, pp. 111–140, 2008.
[27] H. Flordal, R. Malik, M. Fabian, and K. Åkesson, “Compositional synthesis of maximally permissive supervisors using supervision equivalence,” Discrete Event Dynamic Syst., vol. 17, no. 4, pp. 475–504, 2007.