Towards an Efficient Tree Automata Based Technique for Timed Systems[*]

S. Akshay[1], Paul Gastin[2], Shankara Narayanan Krishna[3], and Ilias Sarkar[4]

1 **Dept of CSE, IIT Bombay, India**
akshayss@cse.iitb.ac.in
2 **LSV, ENS Paris-Saclay, CNRS, France**
paul.gastin@lsv.fr
3 **Dept of CSE, IIT Bombay, India**
krishnas@cse.iitb.ac.in
4 **Dept of CSE, IIT Bombay, India**
ilias@cse.iitb.ac.in

Abstract

The focus of this paper is the analysis of real-time systems with recursion, through the development of good theoretical techniques which are implementable. Time is modeled using clock variables, and recursion using stacks. Our technique consists of modeling the behaviours of the timed system as graphs, and interpreting these graphs on tree terms by showing a bound on their tree-width. We then build a tree automaton that accepts exactly those tree terms that describe realizable runs of the timed system. The emptiness of the timed system thus boils down to emptiness of a finite tree automaton that accepts these tree terms. This approach helps us in obtaining an optimal complexity, not just in theory (as done in earlier work e.g. [4]), but also in going towards an efficient implementation of our technique. To do this, we make several improvements in the theory and exploit these to build a first prototype tool that can analyze timed systems with recursion.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Timed automata, tree automata, pushdown systems, tree-width

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.39

1 Introduction

Development of efficient techniques for the verification of real time systems is a practically relevant problem. Timed automata [6] are a prominent and well accepted abstraction of timed systems. The development of this model originally began with highly theoretical results, starting from the PSPACE-decision procedure for the emptiness of timed automata. But later, this theory has led to the development of state of the art and industrial strength tools like UPPAAL [7]. Currently, such tools are being adapted to build prototypes that handle other systems such timed games, stochastic timed systems etc. While this helps in analysis of certain systems, there are complicated real life examples that require paradigms like recursion, multi-threaded concurrency and so on.

[*] This work was partly supported by UMI-ReLaX, DST-CEFIPRA project AVeRTS and DST-INSPIRE faculty award [IFA12-MA-17].

For timed systems with recursion, a popular theoretical framework is the model of timed pushdown automata (TPDA). In this model, in addition to clock variables as in timed automata, a stack is used to model recursion. Depending on how clocks and stack operations are integrated, several variants [8], [1], [14], [12], [9] have been looked at. For many of these variants, the basic problem of checking emptiness has been shown decidable (and EXPTIME-complete) using different techniques. The proofs in [8], [1], [14] work by adapting the technique of region abstraction to untime the stack and obtain a usual untimed pushdown automaton, while [9] gives a proof by reasoning with sets of timed atoms. Recently, in [4], a new proof technique was introduced which modeled the behaviours of the TPDA as graphs with timing constraints and analyzed these infinite collections of time-constrained graphs using tree automata. This approach follows the template which has been explored in depth for various untimed systems in [13], [11], [3]. The basic idea can be outlined as follows: (1) describe behaviours of the underlying system as graphs, (2) show that this class of graphs has bounded width, (3) either appeal to Courcelle's theorem [10] by showing that the desired properties are MSO-defineable or explicitly construct a tree-automaton to capture the class of graphs that are the desired behaviours. The work in [4] extends this approach to timed systems, by considering their behaviors as time-constrained words. The main difficulty here is to obtain a tree automaton that accepts only those time-constrained words that are *realizable* via a valid time-stamping.

Despite the amount of theoretical work in this area [8, 13, 11, 4, 1, 9], none of these algorithms have been implemented to the best of our knowledge. Applying Courcelle's theorem is known to involve a blowup in the complexity (depending on the quantifier-alternation of the MSO formula). The algorithm for checking emptiness in [4] for the timed setting which directly constructs the tree automaton avoiding the MSO translation also turns out to be unimplementable even for small examples due to the following reasons: First, it has a pre-processing step where each transition in the underlying automaton is broken into several micro transitions, one for each constraint that is checked there, and one corresponding to each clock that gets reset on that transition. This results in a blowup in the size of the automaton. Second, the number of states of the tree automaton that is built to check realizabilty as well as the existence of a run of a system is bounded by $(M \times T)^{\mathcal{O}(K^2)} 2^{\mathcal{O}(K^2 lg K)}$, where M is one more than the maximal constant used in the given system, T is the number of transitions, and $K = 4|X| + 6$ is the so-called split-width, where $|X|$ is the number of clocks used. This implies that even for a system that has 1 clock, 5 transitions and uses a maximum constant 5, we have more than 30^{100} states.

In this paper, we take the first steps towards an efficient implementation. While we broadly follow the graph and tree-automata based approach (and in particular [4]), our main contribution is to give an efficient technique for analyzing TPDA. This requires several fundamental advances: (i) we avoid the preprocessing step, obtaining a direct bound on tree width for timed automata and TPDA. This is established by playing a *split-game* which decomposes the graph representing behaviours of the timed system into tree terms; by coloring some vertices of the graph and removing certain edges whose endpoints are colored. The minimum number of colors used in a winning strategy is 1 plus the tree-width of the graph. (ii) we develop a new algorithm for building the tree automaton for emptiness, whose complexity is in ETIME, i.e., bounded by $(M \times T)^{3|X|+3}$ with an exponent which is a linear function of the input size (improved from EXPTIME, where the exponent is a polynomial function of the input). Thus, if the system has 1 clock, 5 transitions and uses a maximum constant 5, we have only $\sim 30^6$ states. In particular, our tree-automaton is *strategy-driven*, i.e., it manipulates only those tree terms that arise out of a winning strategy of our split-game. As a result of this strategy-guided approach, the number of states of our tree automaton is highly optimized, and an accepting run exactly corresponds to the moves in a winning strategy of our split-game. (iii) Finally, our algorithm outputs a witness for realizability (and non-emptiness). As a proof-of-concept, we implemented our algorithm and despite the worst-case complexity, we discuss optimizations, results and a modeling example where our implementation performs well.

Due to lack of space, we have not included all the proofs here; the full version of the paper, with proofs of all the results, illustrative examples, details of experimental results and benchmarks used can be found at [5].

2 Graphs for behaviors of timed systems

We fix an alphabet Σ and use Σ_ε to denote $\Sigma \cup \{\varepsilon\}$, where ε is the silent action. We also fix a finite set of intervals $\mathcal{I}$ with bounds in $\mathbb{N} \cup \{\infty\}$. For a set S, we use $\leq \subseteq S \times S$ to denote a partial or total order on S. For any $x, y \in S$, we write $x < y$ if $x \leq y$ and $x \neq y$, and $x \lessdot y$ if $x < y$ and there does not exist $z \in S$ such that $x < z < y$.

▶ **Definition 1.** A *word with timing constraints* (TCW) over $(\Sigma, \mathcal{I})$ is a structure $\mathcal{V}=(V, \rightarrow, \lambda, (\curvearrowright^I)_{I\in\mathcal{I}})$ where V is a finite set of vertices or positions, $\lambda \colon V \rightarrow \Sigma_\varepsilon$ labels each position, the reflexive transitive closure $\leq \, = \, \rightarrow^*$ is a total order on V and $\rightarrow \, = \, \lessdot$ is the successor relation, while $\curvearrowright^I \subseteq <$ connects pairs of positions carrying a timing constraint, given by an interval in $I \in \mathcal{I}$. A TCW $\mathcal{V}=(V, \rightarrow, \lambda, (\curvearrowright^I)_{I\in\mathcal{I}})$ is called *realizable* if there exists a timestamp map $\mathsf{ts} \colon V \rightarrow \mathbb{R}_+$ such that $\mathsf{ts}(i) \leq \mathsf{ts}(j)$ for all $i \leq j$ (time is non-decreasing) and $\mathsf{ts}(j) - \mathsf{ts}(i) \in I$ for all $i \curvearrowright^I j$ (timing constraints are satisfied).

An example of a TCW is given in Figure 1 (right), with positions $0, 1, 2, 3$ labelled by $\Sigma = \{a, b, c\}$. Curved edges decorated with intervals connect positions related by $\curvearrowright^I$, while straight edges define the successor relation $\rightarrow$. This TCW is realizable by the sequence of timestamps $0, 0.9, 2.89, 3.1$ but not by $0, 0.9, 2.99, 3.1$. We let $\mathsf{Real}(\Sigma, \mathcal{I})$ be the set of TCWs over $(\Sigma, \mathcal{I})$ which are realizable.

TPDA and their semantics as TCWs. Dense-timed pushdown automata (TPDA), introduced in [1], are an extension of timed automata, and operate on a finite set of real-valued clocks and a stack which holds symbols with their ages. The age of a symbol represents the time elapsed since it was pushed onto the stack. Formally, a TPDA $\mathcal{S}$ is a tuple $(S, s_0, \Sigma, \Lambda, \Delta, X, F)$ where S is a finite set of states, $s_0 \in S$ is the initial state, Σ, Λ, are respectively finite sets of input, stack symbols, Δ is a finite set of transitions, X is a finite set of real-valued variables called clocks, $F \subseteq S$ are final states. A transition $t \in \Delta$ is a tuple $(s, \gamma, a, \mathsf{op}, R, s')$ where $s, s' \in S$, $a \in \Sigma$, γ is a finite conjunction of atomic formulae of the kind $x \in I$ for $x \in X$ and $I \in \mathcal{I}$, $R \subseteq X$ are the clocks reset, op is one of the following stack operations:

1. nop does not change the contents of the stack,
2. $\downarrow_c$, $c \in \Lambda$ is a push operation that adds c on top of the stack, with age 0.
3. $\uparrow_c^I$, $c \in \Lambda$ is a stack symbol and $I \in \mathcal{I}$ is an interval, is a pop operation that removes the top most symbol of the stack provided it is a c with age in the interval I.

Timed automata (TA) can be seen as TPDA using nop operations only. This definition of TPDA is equivalent to the one in [1], but allows checking conjunctive constraints and stack operations together. In [9], it is shown that TPDA of [1] are expressively equivalent to timed automata with an untimed stack. As our technique is oblivious to whether the stack is timed or not, we focus on the syntactically more succinct model TPDA with a timed stack.

Next, we define the semantics of a TPDA in terms of TCWs.

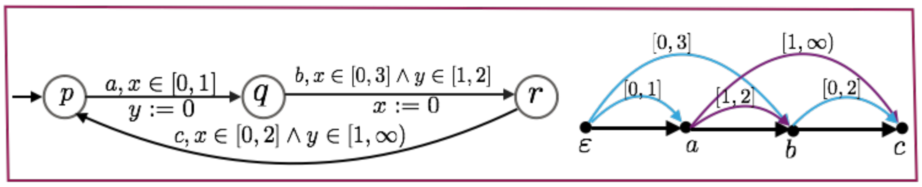


Figure 1 A timed automaton and a TCW capturing a run.

▶ **Definition 2.** A TCW $\mathcal{V} = (V, \rightarrow, \lambda, (\curvearrowright^I)_{I \in \mathcal{I}})$ is said to be *generated or accepted* by a TPDA $\mathcal{S}$ if there is an accepting abstract run $\rho = (s_0, \gamma_1, a_1, \mathsf{op}_1, R_1, s_1)$ $(s_1, \gamma_2, a_2, \mathsf{op}_2, R_2, s_2) \cdots (s_{n-1}, \gamma_n, a_n, \mathsf{op}_n, R_n, s_n)$ of $\mathcal{S}$ such that, $s_n \in F$ and

- the sequence of push-pop operations is well-nested: in each prefix $\mathsf{op}_1 \cdots \mathsf{op}_k$ with $1 \leq k \leq n$, number of pops is at most number of pushes, and in the full sequence $\mathsf{op}_1 \cdots \mathsf{op}_n$, they are equal; and
- $V = \{0, 1, \dots, n\}$ with $\lambda(0) = \varepsilon$ and $\lambda(i) = a_i$ for all $1 \leq i \leq n$ and $0 \rightarrow 1 \rightarrow \cdots \rightarrow n$ and, for all $I \in \mathcal{I}$, the relation $\curvearrowright^I$ is the set of pairs (i, j) with $0 \leq i < j \leq n$ such that
  - either for some $x \in X$ we have $x \in R_i$ (assuming $R_0 = X$) and $x \in I$ is a conjunct of γ_j and $x \notin R_k$ for all $i < k < j$,
  - or $\mathsf{op}_i = \downarrow_b$ is a push and $\mathsf{op}_j = \uparrow_b^I$ is the matching pop (same number of pushes and pops in $\mathsf{op}_{i+1} \cdots \mathsf{op}_{j-1}$).

We denote by $\mathsf{TCW}(\mathcal{S})$ the set of TCWs generated by $\mathcal{S}$. The non-emptiness problem for the TPDA $\mathcal{S}$ amounts to asking whether some TCW generated by $\mathcal{S}$ is realizable, i.e., whether $\mathsf{TCW}(\mathcal{S}) \cap \mathsf{Real}(\Sigma, \mathcal{I}) \neq \emptyset$. The TCW semantics of timed automata (TA) can be obtained from the above discussion by just ignoring the stack components (using nop operations only). Figure 1 depicts a simple example of a timed automaton and a TCW generated by it.

▶ Remark. The classical semantics of timed systems is given in terms of timed words. A *timed word* is a sequence $w = (a_1, t_1) \cdots (a_n, t_n)$ with $a_1, \dots, a_n \in \Sigma$ and $(t_i)_{1 \leq i \leq n}$ is a non-decreasing sequence of values in $\mathbb{R}_+$. A *realization* of a TCW $\mathcal{V} = (V, \rightarrow, \lambda, (\curvearrowright^I)_{I \in \mathcal{I}}) \in \mathsf{TCW}(\mathcal{S})$ with $V = \{0, 1, \dots, n\}$ is a timed word $w = (\lambda(1), \mathsf{ts}(1)) \dots (\lambda(n), \mathsf{ts}(n))$ where the timestamp map $\mathsf{ts}\colon V \rightarrow \mathbb{R}_+$ (with $\mathsf{ts}(0) = 0$) is non decreasing and satisfies all timing constraints of $\mathcal{V}$. For example, the timed word $(a, 0.9)(b, 2.89)(c, 3.1)$ is a realization of the TCW in Figure 1 while $(a, 0.9)(b, 2.99)(c, 3.1)$ is not. It is not difficult to check that the language $\mathcal{L}(\mathcal{S})$ of timed words accepted by $\mathcal{S}$ with the classical semantics is precisely the set of realizations of TCWs in $\mathsf{TCW}(\mathcal{S})$. Therefore, $\mathcal{L}(\mathcal{S}) = \emptyset$ iff $\mathsf{TCW}(\mathcal{S}) \cap \mathsf{Real}(\Sigma, \mathcal{I}) = \emptyset$.

We now identify some important properties satisfied by TCWs generated from a TPDA. Let $\mathcal{V} = (V, \rightarrow, \lambda, (\curvearrowright^I)_{I \in \mathcal{I}})$ be a TCW. The matching relation $(\curvearrowright^I)_{I \in \mathcal{I}}$ is used in two contexts: (i) while connecting a clock reset point (say for clock x) to a point where a guard of the form $x \in I$ is checked, and (ii) while connecting a point where a push was made to its corresponding pop, where the age of the topmost stack symbol is checked to be in interval I. We use the notations $\curvearrowright^{x \in I}$ and $\curvearrowright^{s \in I}$ to denote the matching relation $\curvearrowright^I$ corresponding to a clock-reset-check as well as push on stack-check respectively. We say that $\mathcal{V}$ is *well timed* w.r.t. a set of clocks X and a stack s if for each interval $I \in \mathcal{I}$ the matching relation $\curvearrowright^I$ can be partitioned as $\curvearrowright^I = \curvearrowright^{s \in I} \uplus \biguplus_{x \in X} \curvearrowright^{x \in I}$ where

(T_1) the stack relation $\curvearrowright^s = \bigcup_{I \in \mathcal{I}} \curvearrowright^{s \in I}$ corresponds to the matching push-pop events, hence it is well-nested: for all $i \curvearrowright^s j$ and $i' \curvearrowright^s j'$, if $i < i' < j$ then $j' < j$.

(T_2) For each $x \in X$, the clock relation $\curvearrowright^x = \bigcup_{I \in \mathcal{I}} \curvearrowright^{x \in I}$ corresponds to the timing constraints for clock x and respects the last reset condition: for all $i \curvearrowright^x j$ and $i' \curvearrowright^x j'$, if $i < i'$, then $j \leq i'$. See Figure 1 for example, where $0 \curvearrowright^x 2$ and $2 \curvearrowright^x 3$.

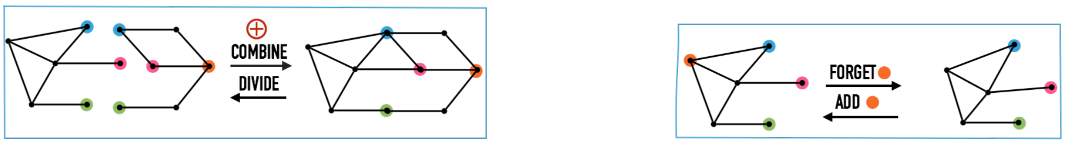


Figure 2 Operations on colored graphs.

It is then easy to check that TCWs defined by a TPDA with set of clocks X are well-timed for the set of clocks X, i.e., satisfy the properties above. We obtain the same for TA by just ignoring the stack edges, i.e., (T_1) above.

3 Tree-Width for Timed Systems

In this section, we discuss tree-algebra by introducing the basic terms, the operations on terms, their syntax and semantics. This will help us in analyzing the graphs obtained in the previous section using tree-terms, and establishing a bound on the tree-width.

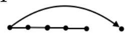
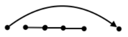
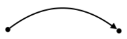
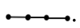
We introduce tree terms TTs from Courcelle [10] and their semantics as graphs which are both vertex-labeled and edge-labeled. Let Σ be a set of vertex labels and let Ξ be a set of edge labels. Let $K \in \mathbb{N}$. The syntax of K-tree terms K-TTs over (Σ, Ξ) is given by

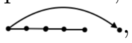
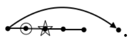
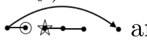
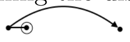
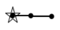
$$\tau ::= (a,i) \mid (a,i)\xi(b,j) \mid \mathsf{Forget}_i\, \tau \mid \mathsf{Rename}_{i,j}\, \tau \mid \tau \oplus \tau$$

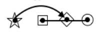
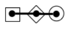
where $i, j \in \{1, 2, \dots, K\}$ are colors $(i \neq j)$, $a, b \in \Sigma$ are vertex labels and $\xi \in \Xi$ is an edge label. The semantics of a K-TT τ is a colored graph $[\![\tau]\!] = (G_\tau, \chi_\tau)$ where $G_\tau = (V, E)$ is a graph and $\chi_\tau \colon \{1, 2, \dots, K\} \to V$ is a partial injective function assigning a color to some vertices of G_τ. Note that any color in $\{1, 2, \dots, K\}$ is assigned to at most one vertex of G_τ.

The atomic term (a, i) is a single vertex colored i and labeled a and the atomic term $(a, i)\xi(b, j)$ represents a ξ-labeled edge between two vertices colored i, j and labeled a, b respectively. Given a tree term τ, $\mathsf{Forget}_i(\tau)$ forgets the color i from a node colored i, leaving it uncolored. The operation $\mathsf{Rename}_{i,j}(\tau)$ renames the color i of a node to color j, provided no nodes are already colored j. Since any color appears at most once in G_τ, the operations $\mathsf{Forget}_i(\tau)$ and $\mathsf{Rename}_{i,j}(\tau)$ are deterministic, when colors i, j, are fixed. Finally, the operation $\tau_1 \oplus \tau_2$ (read as combine) combines two terms τ_1, τ_2 by fusing the nodes of τ_1, τ_2 which have the same color. See Figure 2.

The tree-width of a graph G is defined as the least K such that $G = G_\tau$ for some TT τ using $K + 1$ colors. Let TW_K denote the set of all graphs having tree width at most K. For TCWs, we have successor edges $\rightarrow$ and matching edges $\curvearrowright^I$ where $I \in \mathcal{I}$ is an interval. Hence, the set of edge labels is $\Xi_{\mathcal{I}} = \{\rightarrow\} \cup \{\curvearrowright^I \mid I \in \mathcal{I}\}$ and we use TTs over $(\Sigma, \Xi_{\mathcal{I}})$. An example is given in [5, Appendix A].

TCWs and Games. We find it convenient to prove that TCWs have bounded tree-width by playing a game, whose game positions are TCWs in which some successor edges may have been cut, i.e., are missing. Such TCWs, where some successor edges may be missing, are called split-TCWs. A split-TCW which is a connected graph is called a connected split-TCW, while a split-TCW which is a disconnected graph, is called a disconnected split-TCW. For example,  is a connected split-TCW, while  is a disconnected split-TCW consisting of two connected split TCWs, namely  and .

A TCW is atomic if it is denoted by an atomic term $((a, i)$ or $(a, i) \rightarrow (b, j)$ or $(a, i) \curvearrowright^I (b, j))$. *The split-game* is a two player turn based game $\mathcal{G} = (\mathsf{Pos}_\exists \uplus \mathsf{Pos}_\forall, \mathsf{Moves})$ where Eve's set of game positions $\mathsf{Pos}_\exists$ consists of all connected (wrt. $\rightarrow \cup \curvearrowright$) split-TCWs and Adam's set of game positions $\mathsf{Pos}_\forall$ consists of dis-connected split-TCWs. Eve's moves consist of adding colors to the vertices of the split-TCW, and dividing the split-TCW. For example, if we have the connected split-TCW , and Eve colors two nodes (we use shapes in place of colors for better visibility) we obtain . This graph can be divided obtaining the disconnected graph  and . As a result, we obtain the connected parts  and  and . Now Adam's choices are on this disconnected split-TCW and he can choose either of the above three connected split-TCWs to continue the game. Thus, divide is the reverse of the combine operation $\oplus$. Adam's moves amount to choosing a connected component of the split-TCW. Eve has to continue coloring and dividing on the connected split-TCW chosen by Adam. Atomic split-TCWs are terminal positions in the game: neither Eve nor Adam can move from an atomic split-TCW. A play on a split-TCW $\mathcal{V}$ is a path in $\mathcal{G}$ starting from $\mathcal{V}$ and leading to an atomic split-TCW. The cost of the play is the maximum width (number of colors-1) of any split-TCW encountered in the path. In our example above, is already an atomic split-TCW. If Adam chooses any of the other two, it is easy to see that Eve has a strategy using at most 2 colors in any of the split-TCWs that will be obtained till termination. The *cost* of a strategy σ for Eve from a split-TCW $\mathcal{V}$ is the maximal cost of the plays starting from $\mathcal{V}$ and following strategy σ. The *tree-width* of a (split-)TCW $\mathcal{V}$ is the minimal cost of Eve's (positional) strategies starting from $\mathcal{V}$. Let TCW_K denote the set of TCWs with tree-width bounded by K.

A *block* in a split-TCW is a maximal set of points of V connected by $\rightarrow$. For example, the split-TCW  has one non-trivial block  and one trivial block . Points that are not left or right endpoints of blocks of $\mathcal{V}$ are called internal.

The Bound. We show that we can find a K such that all the behaviors of the given timed system have tree-width bounded by K.

▶ **Theorem 3.** *Given a timed system $\mathcal{S}$ using a set of clocks X, all graphs in its* TCW *language have tree-width bounded by K, i.e.,* $\mathsf{TCW}(\mathcal{S}) \subseteq \mathsf{TCW}_K$*, where*
1. $K = |X| + 1$ *if $\mathcal{S}$ is a timed automaton,*
2. $K = 3|X| + 2$ *if $\mathcal{S}$ is a timed pushdown automaton.*

The following lemma completes the proof of Theorem 3 (2).

▶ **Lemma 4.** *The tree-width of a well-timed* TCW *is bounded by* $3|X| + 2$.

We prove this by playing the "split game" between *Adam* and *Eve* in which *Eve* has a strategy to disconnect the word without introducing more than $3|X| + 3$ colors. *Eve*'s strategy processes the word from right to left. Starting from any TCW, Eve colors the end points of the TCW, as well as the last reset points (from the right end) corresponding to each clock. Here she uses at most $|X| + 2$ colors. On top of this, depending on the last point, we have different cases, as sketched below (a detailed proof is in [5, Appendix B]).

If the last point is the target of a $\curvearrowright^x$ edge for some clock x, then Eve simply removes the clock edge, since both the source and target points of this edge are colored. We only discuss in some detail the case when the last point is the target of a $\curvearrowright^s$ edge, and the source of this edge is an internal point in the non-trivial block. Figure 3 illustrates this case.

To keep a bound on the number of colors needed, Eve divides the TCW as follows:
- First Eve adds a color to the source of the stack edge
- If there are any clock edges crossing this stack edge, Eve adds colors to the corresponding reset points. Note that this results in adding atmost $|X|$ colors.

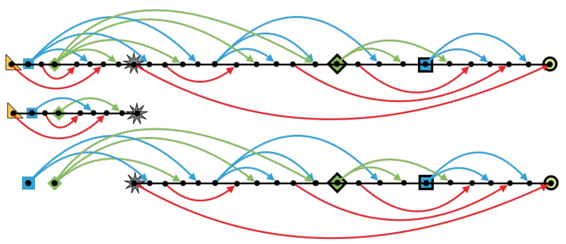


Figure 3 The last point is the target of a $\curvearrowright^s$ (top figure). After the split, we obtain the words $\mathcal{V}_1$ (the middle one) and $\mathcal{V}_2$ (the bottom one).

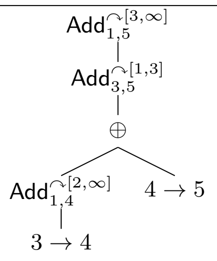
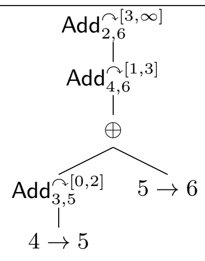
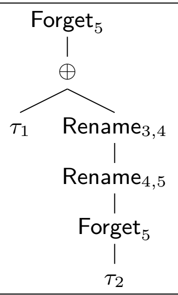
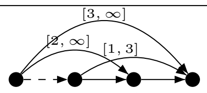
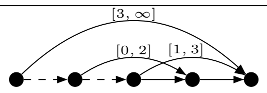
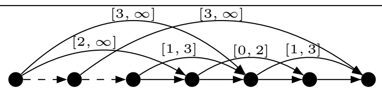
- Eve disconnects the TCW into two parts, such that the right part $\mathcal{V}_2$ consists of one non-trivial block whose end points are the source and target points of the stack edge, and also contains to the left of this block, atmost $|X|$ trivial blocks. Each of these trivial blocks are the reset points of those clock edges which cross over. The left part $\mathcal{V}_1$ is a TCW consisting of all points to the left of the source of the stack edge, and has all remaining edges other than the clock edges which have crossed over. Adam can now continue the game choosing $\mathcal{V}_1$ or $\mathcal{V}_2$. Note that in one of the words so obtained, the stack edge completely spans the non-trivial block, and can be easily removed.

Invariants and bound on tree-width. We now discuss some invariants on the structure of the split-TCWs as we play the game using the above strategy.
(I1) We have $\leq |X|$ colored trivial blocks to the left of the only non-trivial block,
(I2) The last reset node of each clock on the non-trivial block is colored,
(I3) The end points of the non-trivial block are colored.
To maintain the above invariants, we need $|X|+1$ extra colors than the at most $2|X|+2$ mentioned above. This proves that the tree-width of a TPDA with set of clocks X is bounded by $3|X|+2$. If the underlying system is a timed automaton, then we have a single non-trivial block in the game at any point of time. There are no trivial blocks, unlike the TPDA, due to the absence of stack edges. This results in using only $\leq |X|+2$ colors at any point of time, where $|X|$ colors are needed to color the last reset points of the clocks in the block, and the 2 colors are used to color the left, right end points of the block.

4 Tree automata for Validity

In this section, we give one of the most challenging constructions (Theorem 6) of the paper, namely, the tree automaton that accepts all valid and realizable K-TTs which are "good". Good K-TTs are defined below. In this section, we restrict ourselves to closed intervals; that is, those of the form $[a, b]$ and $[a, \infty)$, where $a, b \in \mathbb{N}$. Fix $K \geq 2$. Not all graphs defined by K-TTs are realizable TCWs. Indeed, if τ is such a TT, the edge relation $\rightarrow$ may have cycles or may be branching, which is not possible in a TCW. Also, the timing constraints given by $\curvearrowright^I$ need not comply with the $\rightarrow$ relation: for instance, we may have a timing constraint $e \curvearrowright^I f$ with $f \rightarrow^+ e$ ($\rightarrow^+$ is the transitive closure of $\rightarrow$, i.e., e can be reached from f after taking ≥ 1 successor edges $\rightarrow$). Moreover, some terms may define graphs denoting TCWs which are not realizable. So we use $\mathcal{A}^{K,M}_{\mathsf{valid}}$ to check for validity. Since we have only closed intervals in timing constraints, integer timestamps suffice for realizability, as can be seen from the following lemma ([5, Appendix C.1]).

Table 1 The second row gives tree representations of three good 6-TTs τ_1, τ_2, τ_3. In all these terms, we ignore vertex labels and we use $\mathsf{Add}_{i,j}^{\curvearrowright I}\,\tau$ as a macro for $\tau \oplus i \curvearrowright^I j$. The third row gives their semantics $[\![\tau]\!] = (G_\tau, \chi_\tau)$ together with a realization ts, the fourth row gives possible states q of $\mathcal{A}_{\mathsf{valid}}^{K,M}$ with $M = 4$ after reading the terms. Here, L is the circled color. The boolean value $\mathsf{acc}(i)$ for each non maximal color i is written between $\mathsf{tsm}(i)$ and $\mathsf{tsm}(i^+)$.

τ_1	τ_2	τ_3
 $\mathsf{Add}_{1,5}^{\curvearrowright[3,\infty]}$ $\mathsf{Add}_{3,5}^{\curvearrowright[1,3]}$ $\oplus$ $\mathsf{Add}_{1,4}^{\curvearrowright[2,\infty]}$ $4 \to 5$ $3 \to 4$	 $\mathsf{Add}_{2,6}^{\curvearrowright[3,\infty]}$ $\mathsf{Add}_{4,6}^{\curvearrowright[1,3]}$ $\oplus$ $\mathsf{Add}_{3,5}^{\curvearrowright[0,2]}$ $5 \to 6$ $4 \to 5$	 Forget_5 $\oplus$ τ_1 $\mathsf{Rename}_{3,4}$ $\mathsf{Rename}_{4,5}$ Forget_5 τ_2
 χ 1 3 4 5 ts 0 5 6 8	 χ 2 3 4 5 6 ts 3 6 8 8 11	 χ 1 2 3 4 6 ts 0 3 5 6 8 8 11
P 1 $\textcircled{3}$ 4 5 tsm 0 ¬acc 1 acc 2 acc 0	P 2 3 $\textcircled{4}$ 5 6 tsm 3 acc 2 acc 0 acc 0 acc 3	P 1 2 $\textcircled{3}$ 4 6 tsm 0 acc 3 acc 1 acc 2 ¬acc 3

▶ **Lemma 5.** *Let $\mathcal{V} = (V, \to, \lambda, (\curvearrowright^I)_{I\in\mathcal{I}(M)})$ be a* TCW *using only closed intervals in its timing constraints. Then, $\mathcal{V}$ is realizable iff there exists an integer valued timestamp map satisfying all timing constraints.*

Consider a set of colors $P \subseteq \{1, \dots, K\}$. For each $i \in P$ we let $i^+ = \min\{j \in P \cup \{\infty\} \mid i < j\}$ and $i^- = \max\{j \in P \cup \{0\} \mid j < i\}$. If P is not clear from the context, then we write $\mathsf{next}_P(i)$ and $\mathsf{prev}_P(i)$. Given a K-TT τ with semantics $[\![\tau]\!] = (G, \chi)$, we denote by $\mathsf{Act} = \mathsf{dom}(\chi)$ the set of active colors in τ, we let $\mathsf{Right} = \max(\mathsf{Act})$ and $\mathsf{Left} = \min\{i \in \mathsf{Act} \mid \chi(i) \to^* \chi(\mathsf{Right})\}$. If τ is not clear from the context, then we write Act_τ, Left_τ and Right_τ. A K-TT τ is *good* if

- $\tau ::= (a, i) \to (b, j) \mid (a, i) \curvearrowright^I (b, j) \mid \mathsf{Forget}_i\,\tau \mid \mathsf{Rename}_{i,j}\,\tau \mid \tau \oplus \tau$,
- for every subterm of the form $(a, i) \to (b, j)$ or $(a, i) \curvearrowright^I (b, j)$ we have $i < j$,
- $\mathsf{Rename}_{i,j}\,\tau$ is possible only if $i^- < j < i^+$,
- $\tau_1 \oplus \tau_2$ is allowed if $\mathsf{Right}_1 = \mathsf{Left}_2$ and $\{i \in \mathsf{Act}_2 \mid \mathsf{Left}_1 \le i \le \mathsf{Right}_1\} \subseteq \mathsf{Act}_1$.

Examples of good TTs and their semantics are given in Table 1. Note that the semantics of a K-TT τ is a colored graph $[\![\tau]\!] = (G_\tau, \chi_\tau)$. Below, we provide a direct construction of a tree automaton, which gives a clear upper bound on the size of $\mathcal{A}_{\mathsf{valid}}^{K,M}$, since obtaining this bound gets very technical if we stick to MSO.

▶ **Theorem 6.** *We can build a tree automaton $\mathcal{A}_{\mathsf{valid}}^{K,M}$ with $M^{\mathcal{O}(K)}$ number of states such that $\mathcal{L}(\mathcal{A}_{\mathsf{valid}}^{K,M})$ is the set of* good *K-TTs τ such that $[\![\tau]\!]$ is a* realizable TCW *and the endpoints of $[\![\tau]\!]$ are the only colored points.*

Proof. The tree automaton $\mathcal{A}_{\mathsf{valid}}^{K,M}$ reads the TT bottom-up and stores in its state a finite abstraction of the associated graph. The finite abstraction will keep only the colored points of the graph. We will only accept *good* terms for which the natural order on the active colors

Table 2 Transitions of $\mathcal{A}^{K,M}_{\mathsf{valid}}$. See Table 1 for some intuitions. $I.up$ in row 2 represents upper bound of interval I.

$(a,i) \to (b,j)$	$\bot \xrightarrow{(a,i)\to(b,j)} q = (P, L, \mathsf{tsm}, \mathsf{acc})$ is a transition if $i < j$ and $P = \{i, j\}$, $L = i$ and $\mathsf{acc}(j) = \mathsf{ff}$. The values for $\mathsf{tsm}(i)$, $\mathsf{tsm}(j)$ and $\mathsf{acc}(i)$ are guessed.
$(a,i) \curvearrowright^I (b,j)$	$\bot \xrightarrow{(a,i)\curvearrowright^I(b,j)} q = (P, L, \mathsf{tsm}, \mathsf{acc})$ is a transition if $i < j$ and $P = \{i, j\}$, $L = j$ and $\mathsf{acc}(j) = \mathsf{ff}$. Here, i and j are trivial blocks. The values for $\mathsf{tsm}(i)$, $\mathsf{tsm}(j)$ and $\mathsf{acc}(i)$ are guessed such that ($\mathsf{acc}(i) = \mathsf{tt}$ and $d(i,j) \in I$) or ($\mathsf{acc}(i) = \mathsf{ff}$ and $I.up = \infty$).
$\mathsf{Rename}_{i,j}$	$q = (P, L, \mathsf{tsm}, \mathsf{acc}) \xrightarrow{\mathsf{Rename}_{i,j}} q' = (P', L', \mathsf{tsm}', \mathsf{acc}')$ is a transition if $i \in P$ and $i^- < j < i^+$. Then, q' is obtained from q by replacing i by j.
Forget_i	$q = (P, L, \mathsf{tsm}, \mathsf{acc}) \xrightarrow{\mathsf{Forget}_i} q' = (P', L', \mathsf{tsm}', \mathsf{acc}')$ is a transition if $L < i < \max(P)$ (endpoints should stay colored). Then, state q' is deterministically given by $P' = P \setminus \{i\}$, $L' = L$, $\mathsf{tsm}' = \mathsf{tsm}_{\restriction P'}$ and $\mathsf{acc}'(i^-) = \mathsf{ACC}(i^-, i^+) \wedge (D(i^-, i^+) < M)$, the other values of acc' are inherited from acc.
$\oplus$	$q_1, q_2 \xrightarrow{\oplus} q$ where $q_1 = (P_1, L_1, \mathsf{tsm}_1, \mathsf{acc}_1)$, $q_2 = (P_2, L_2, \mathsf{tsm}_2, \mathsf{acc}_2)$ and $q = (P, L, \mathsf{tsm}, \mathsf{acc})$ is a transition if the following hold ▪ $R_1 = \max(P_1) = L_2$ and $\{i \in P_2 \mid L_1 \le i \le R_1\} \subseteq P_1$ (we cannot insert a new point from the second argument in the non-trivial block of the first argument). ▪ $P = P_1 \cup P_2$, $L = L_1$, and $\mathsf{tsm}_{\restriction P_1} = \mathsf{tsm}_1$ and $\mathsf{tsm}_{\restriction P_2} = \mathsf{tsm}_2$: these updates are deterministic. In particular, this implies that tsm_1 and tsm_2 coincide on $P_1 \cap P_2$. ▪ Finally, acc satisfies $\mathsf{acc}(\max(P)) = \mathsf{ff}$ and $\forall i \in P_1 \setminus \{\max(P_1)\}$ $\quad \mathsf{acc}_1(i) \iff \mathsf{ACC}_q(i, \mathsf{next}_{P_1}(i)) \wedge D_q(i, \mathsf{next}_{P_1}(i)) < M$ $\forall i \in P_2 \setminus \{\max(P_2)\}$ $\quad \mathsf{acc}_2(i) \iff \mathsf{ACC}_q(i, \mathsf{next}_{P_2}(i)) \wedge D_q(i, \mathsf{next}_{P_2}(i)) < M$. Notice that these conditions imply For all $i \in P_1$, if $\mathsf{next}_P(i) = \mathsf{next}_{P_1}(i)$ (e.g., if $L_1 \le i < R_1$) then $\mathsf{acc}(i) = \mathsf{acc}_1(i)$. For all $i \in P_2$, if $\mathsf{next}_P(i) = \mathsf{next}_{P_2}(i)$ (e.g., if $L_2 \le i$) then $\mathsf{acc}(i) = \mathsf{acc}_2(i)$.

coincides with the order of the corresponding vertices in the final TCW. The restriction to good terms ensures that the graph defined by the TT is a split-TCW.

Moreover, to ensure realizability of the TCW defined by a term, we will guess timestamps of vertices modulo M. We also guess while reading a subterm whether the time elapsed between two consecutive active colors is *big* ($\geq M$) or *small* ($< M$). We see below that the time elapsed is *small* iff it can be recovered accurately with the modulo M abstraction. Then, the automaton has to check that all these guesses are coherent and using these values it will check that every timing constraint is satisfied.

Formally, *states of* $\mathcal{A}^{K,M}_{\mathsf{valid}}$ are tuples of the form $q = (P, L, \mathsf{tsm}, \mathsf{acc})$, where $P \subseteq \{1, \ldots, K\}$, $L \in P$, $\mathsf{tsm}\colon P \to [M] = \{0, \ldots, M-1\}$ and $\mathsf{acc}\colon P \to \mathbb{B}$. acc is a flag which stands for "accurate", and is used to check if the time elapse between two points is accurate or not, based on the time stamps.

Intuitively, when reading bottom-up a K-TT τ with $[\![\tau]\!] = (V, \to, \lambda, (\curvearrowright^I)_{I\in\mathcal{I}}, \chi)$, the automaton $\mathcal{A}^{K,M}_{\mathsf{valid}}$ will reach a state $q = (P, L, \mathsf{tsm}, \mathsf{acc})$ such that

(A$_1$) $P = \mathsf{Act}$ is the set of *active* colors in τ, $L = \mathsf{Left}$ and $\max(P) = \mathsf{Right}$.

(A$_2$) For all $i \in P$, if $L \le i < \max(P)$ then $\chi(i) \to^+ \chi(i^+)$ in $[\![\tau]\!]$.

(A$_3$) Let $\dashrightarrow = \{(\chi(i), \chi(i^+)) \mid i \in P \wedge i < L\}$. This extra relation serves at ordering the blocks of a split-TCW. Then, $([\![\tau]\!], \dashrightarrow)$ is an *ordered* split-TCW, i.e., $< = (\to \cup \dashrightarrow)^+$ is a total order on V, timing constraints in $[\![\tau]\!]$ are $<$-compatible $\curvearrowright^I \subseteq <$ for all I, the *direct successor* relation of $<$ is $\lessdot = \to \cup \dashrightarrow$ and $\to \cap \dashrightarrow = \emptyset$. Moreover, targets of timing constraints are in the last block: for all $u \curvearrowright^I v$ in $([\![\tau]\!], \dashrightarrow)$, we have $\chi(L) \to^* v$.

(A_4) There exists a timestamp map $\mathsf{ts}\colon V \to \mathbb{N}$ such that

- all constraints are satisfied: $\mathsf{ts}(v) - \mathsf{ts}(u) \in I$ for all $u \curvearrowright^I v$ in $[\![\tau]\!]$,
- time is non-decreasing: $\mathsf{ts}(u) \leq \mathsf{ts}(v)$ for all $u \leq v$,
- $(\mathsf{tsm}, \mathsf{acc})$ is the modulo M abstraction of ts: $\forall i \in P$ we have $\mathsf{tsm}(i) = \mathsf{ts}(\chi(i))[M]$ and $\mathsf{acc}(i) = \mathsf{tt}$ iff $i^+ \neq \infty$ and $\mathsf{ts}(\chi(i^+)) - \mathsf{ts}(\chi(i)) < M$.

We say that the state q is a *realizable abstraction* of a term τ if it satisfies conditions (A_1)–(A_4).

Indeed, the finite state automaton $\mathcal{A}^{K,M}_{\mathsf{valid}}$ cannot store the timestamp map ts witnessing realizability. Instead, it stores the modulo M abstraction $(\mathsf{tsm}, \mathsf{acc})$. We will see that $\mathcal{A}^{K,M}_{\mathsf{valid}}$ can check realizability based on the abstraction $(\mathsf{tsm}, \mathsf{acc})$ of ts and can maintain this abstraction while reading the term bottom-up.

We introduce some notations. Let $q = (P, L, \mathsf{tsm}, \mathsf{acc})$ be a state and let $i, j \in P$ with $i \leq j$. We define $d(i,j) = (\mathsf{tsm}(j) - \mathsf{tsm}(i))[M]$ and $D(i,j) = \sum_{k \in P \mid i \leq k < j} d(k, k^+)$. We also define $\mathsf{ACC}(i,j) = \bigwedge_{k \in P \mid i \leq k < j} \mathsf{acc}(k)$. If the state is not clear from the context, then we write $d_q(i,j)$, $D_q(i,j)$, $\mathsf{ACC}_q(i,j)$. For instance, with the state q_3 corresponding to the term τ_3 of Table 1, we have $\mathsf{ACC}(1,4) = \mathsf{tt}$, $d(1,4) = 2$ and $D(1,4) = 6 = \mathsf{ts}(4) - \mathsf{ts}(1)$ is the $\mathsf{accurate}$ value of the time elapsed. Whereas, $\mathsf{ACC}(3,6) = \mathsf{ff}$ and $d(3,6) = 2 = D(3,6)$ are both *strict modulo-M under-approximations* of the time elapsed $\mathsf{ts}(6) - \mathsf{ts}(3) = 6$. The *transitions of $\mathcal{A}^{K,M}_{\mathsf{valid}}$* are defined in Table 2.

Accepting condition. The accepting states of $\mathcal{A}^{K,M}_{\mathsf{valid}}$ should correspond to abstractions of TCWs. Hence the accepting states are of the form $(\{i,j\}, L, \mathsf{tsm}, \mathsf{acc})$ with $i, j \in \{1, \ldots, K\}$, $i < j$, $L = i$ and $\mathsf{acc}(j) = \mathsf{ff}$. The correctness of this construction is in [5, Appendix C.2], and is obtained by proving (i) the transitions of $\mathcal{A}^{K,M}_{\mathsf{valid}}$ indeed preserve the conditions (A_1)–(A_4), (ii) (A_1)–(A_4) ensure among other things, that the boolean values $\mathsf{acc}(i)$, $\mathsf{ACC}(i,j)$ for $i < j$ indeed defines when the elapse of time is accurately captured by the modulo M abstraction: that is, $\mathsf{ACC}(i,j)$ is true iff the actual time elapse between i and j is captured using the modulo M abstraction $D(i,j)$. ◀

5 Tree automata for timed systems

The goal of this section is to build a tree automaton which accepts tree terms denoting TCWs accepted by a TPDA. The existence of a tree automaton can be proved by showing the MSO definability of the runs of the TPDA $\mathcal{S}$ on a TCW. However, as seen in section 4, we directly construct a tree automaton for better complexity. Given the timed system $\mathcal{S}$, let K be the bound on tree-width given by Theorem 3 and let M be one more than the maximal constant occurring in the guards of $\mathcal{S}$. The automaton $\mathcal{A}^{K,M}_{\mathcal{S}}$ will accept *good* K-TTs with the additional restriction that a timing constraint is immediately combined with an existing term. That is, *restricted* K-TTs are *good* K-TTs restricted to the following syntax:

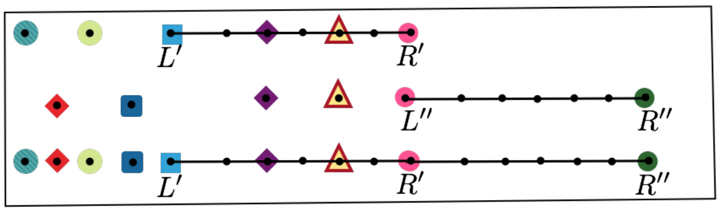
$$\tau ::= (a,i) \to (b,j) \mid \tau \oplus [(a,i) \curvearrowright^I (b,j)] \mid \mathsf{Forget}_i\, \tau \mid \mathsf{Rename}_{i,j}\, \tau \mid \tau \oplus \tau$$

▶ **Theorem 7.** *Let $\mathcal{S}$ be a TPDA of size $|\mathcal{S}|$ (constants encoded in unary) with set of clocks X and using constants less than M. Let K be the bound on tree-width given by Theorem 3. Then, we can build a tree automaton $\mathcal{A}^{K,M}_{\mathcal{S}}$ with $|\mathcal{S}|^{\mathcal{O}(K)} \cdot K^{\mathcal{O}(|X|)}$ states such that $\mathcal{A}^{K,M}_{\mathcal{S}}$ accepts the set of* restricted K-TTs *τ such that $[\![\tau]\!] \in \mathsf{TCW}(\mathcal{S})$. Further, $\mathsf{TCW}(\mathcal{S}) = [\![\mathcal{L}(\mathcal{A}^{K,M}_{\mathcal{S}})]\!] = \{[\![\tau]\!] \mid \tau \in \mathcal{L}(\mathcal{A}^{K,M}_{\mathcal{S}})\}$.*

Proof (Sketch). A state of $\mathcal{A}_S^{K,M}$ is a tuple $q = (P, L, \delta, \mathsf{Push}, \mathsf{Pop}, G, Z)$ where,

- P is the set of active colors, and $L = \mathsf{Left} \in P$ is the left-most point that is connected to the right-end-point $R = \mathsf{Right} = \max(P)$ by successor edges on the non-trivial block.
- δ is a map that assigns to each color $k \in P$ the transition $\delta(k)$ guessed at the leaf corresponding to color k,
- Push and Pop are two boolean variables: $\mathsf{Push} = 1$ iff a push-pop edge has been added to L and $\mathsf{Pop} = 1$ iff a push-pop edge has been added to R,
- $G = (G_x)_{x \in X}$ is a boolean vector of size $|X|$: for each clock $x \in X$, $G_x = 1$ iff some constraint on x has already been checked at R,
- $Z = (Z_x)_{x \in X}$ assigns to each clock x either the color $i \in P$ with $i < L$ of the unique point on the left of the non-trivial block which is the source of a timing constraint $i \curvearrowright^I j$ for clock x, or $\perp$ if no such points exist.

For $j \in P$, let $\mathsf{Reset}(j)$ be the set of clocks that are reset in the transition $\delta(j)$. We describe here the most involved kind of transition $q' \oplus q''$ for states q', q''. The remaining transitions as well as the full proof are in [5, Appendix D]. Let $q' = (P', L', \delta', \mathsf{Push}', \mathsf{Pop}', G', Z')$, $q'' = (P'', L'', \delta'', \mathsf{Push}'', \mathsf{Pop}'', G'', Z'')$ and $q = (P, L, \delta, \mathsf{Push}, \mathsf{Pop}, G, Z)$. Then $q', q'' \xrightarrow{\oplus} q$ is a transition if the following hold:



C$_1$: $R' = \max(P') = L''$ and $\{i \in P'' \mid L' \le i \le R'\} \subseteq P'$ (we cannot insert a new point from the second argument in the non-trivial block of the first argument). Note that according to C$_1$, the points ◆, ⚠ and 🔴 in P'' lying between L', R' are already points in the non-trivial block connecting L' to R'.

C$_2$: $\forall i \in P' \cap P''$, $\delta'(i) = \delta''(i)$ (the guessed transitions match). By C$_2$, the transitions δ', δ'' of ◆, ⚠ and 🔴 must match.

C$_3$: if there is a Push operation in $\delta''(L'')$ then $\mathsf{Push}'' = 1$ and if there is a pop operation in $\delta'(R')$ then $\mathsf{Pop}' = 1$ (the push-pop edges corresponding to the merging point have been added, if they exist). By C$_3$, if $\delta(R') = \delta(L'')$ contains a pop (resp. push) operation then $R' = L''$ is the target (resp. source) of a push-pop edge.

C$_4$: if some guard $x \in I$ is in $\delta(R')$, then $G'_x = 1$ (before we merge, we ensure that the clock guard for x in the transition guessed at R', if any, has been checked). After the merge, $R' = L''$ becomes an internal point; hence by C$_4$, any guard $x \in I$ in $\delta'(R')$ must be checked already, i.e., $G'_x = 1$. After the merge, it is no more possible to add an edge $\curvearrowright^I$ leading into R'.

C$_5$: if $Z'_x \neq \perp$, then $\forall j \in P''$, $Z'_x < j < L'$ implies $x \notin \mathit{Reset}''(j)$ (If a matching edge starting at $Z'_x < L'$ had been seen earlier in run leading to q', then x should not have been reset in q'' between Z'_x and L', else it would violate the consistency of clocks). By C$_5$, if Z'_x is 🔵 (resp. 🟡), i.e., 🔵 (resp. 🟡) is the source of a timing constraint $\curvearrowright^I$ for clock x whose target is in the L'–R' block, then clock x cannot be reset at ♦ and ■ (resp. ■).

C$_6$: if $Z''_x \neq \perp$, then $\forall j \in P'$, $Z''_x < j < L''$ implies $x \notin \mathit{Reset}'(j)$ (If a matching edge starting at $Z''_x < L''$ had been seen earlier in run leading to q'', then x should not have been reset in q' between Z''_x and L''). By C$_6$, if Z''_x is ♦, then x cannot be reset at 🟡, 🟦, ◆, or ⚠. Likewise, if Z''_x was ■, then clock x cannot be reset at 🟦, ◆, or ⚠.

C_7: $P = P' \cup P''$, $L = L'$, $\delta = \delta' \cup \delta''$, $\mathsf{Push} = \mathsf{Push}'$, $\mathsf{Pop} = \mathsf{Pop}''$, $G = G''$ and for all $x \in X$ we have $Z_x = Z''_x$ if $Z''_x < L'$, else $Z_x = Z'_x$. C_7 says that on merging, we obtain the third split-TCW. After the merge, if Z_x is defined, it must be on the left of L', i.e., one of 🔵, ♦, 🟡, 🟦. Notice that the above three conditions ensure the well-nestedness of clocks. By C_5 and C_6 we cannot have both $Z'_x \in \{$🔵, 🟡$\}$ and $Z''_x \in \{$♦, 🟦$\}$. So if $Z''_x \in \{$♦, 🟦$\}$ then $Z_x = Z''_x$ and otherwise $Z_x = Z'_x$ (including when $Z''_x \in \{$◆, ⚠$\}$ and $Z'_x = \perp$).

Accepting Condition. A state $q = (P, L, \delta, \mathsf{Push}, \mathsf{Pop}, G, Z)$ is accepting if $L = \min(P)$, $\delta(L)$ is some dummy ε-transition resetting all clocks and leading to the initial state, $\mathsf{target}(\delta(R))$ is a final state and if $\delta(R)$ has a pop operation then $\mathsf{Pop} = 1$, if it has a constraint/guard for clock x, then $G_x = 1$. Note that the above automaton only accepts restricted K-TTs; this is sufficient for emptiness checking since Eve's winning strategy in Section 3 captures all behaviours of the $\mathsf{TCW}(\mathcal{S})$ while generating only restricted K-TTs. ◀

As a corollary we obtain (see [5, Appendix D.2]),

▶ **Theorem 8.** *Let $\mathcal{S}$ be a TPDA. We have $L(\mathcal{S}) \neq \emptyset$ iff $L(\mathcal{A}^{K,M}_{\mathsf{valid}} \cap \mathcal{A}^{K,M}_{\mathcal{S}}) \neq \emptyset$.*

If the underlying system is a timed automaton, we can restrict the state space to storing just the tuple (P, δ, G) as the other components are not required and L is always $\min(P)$.

Possible Extensions. We now briefly explain how to extend our technique in the presence of diagonal guards: these are guards of the form $x - y \in I$ or $x - \mathsf{pop} \in I$ or $\mathsf{pop} - x \in I$ where x, y are clocks, and I is a time interval. The former is a guard that checks the difference between two clock values, while the latter checks the difference between the value of a clock and the age of the topmost stack symbol at the time of the pop. To handle a constraint of the form $x - y \in I$, it is enough to check the difference between the guessed time stamps at the last reset points of clocks y, x to be in I. Likewise, to check $x - \mathsf{pop} \in I$ or $\mathsf{pop} - x \in I$, we check the difference between the guessed time stamps at the points where the top symbol was pushed on the stack and the last reset of clock x. Note that the last reset points of x, y will not be forgotten until the automaton decides to accept; likewise, the push point will not be forgotten until the pop transition is encountered. Given this, our construction of the tree automaton can be extended with the above checks to handle diagonal guards as well.

6 Implementation and a case-study

We have implemented the emptiness checking procedure for TPDA using our tree-automata based approach, and describe some results here. Despite the $\mathsf{EXPTIME}$-completeness of this problem for general TPDA, we present some good performance results for certain interesting subclasses of TPDA. As a concrete subclass, the complexity significantly improves when there is no extra clock other than the timing constraints associated with the stack; while popping a symbol, we simply check the time elapsed since the push. Note that this can be used to model systems where timing constraints are well-nested: clock resets correspond to push and checking guards corresponds to checking the age of the topmost stack symbol. Thus, this gives a technique for reducing the number of clocks for a timed system with nested timing constraints. For this subclass, the exact number of states of the tree automaton can be improved to $2 \times (M \times T)^2$, where M is 1 plus the maximum constant, and T is the number of transitions. This idea can be extended further to incorporate clocks whose constraints

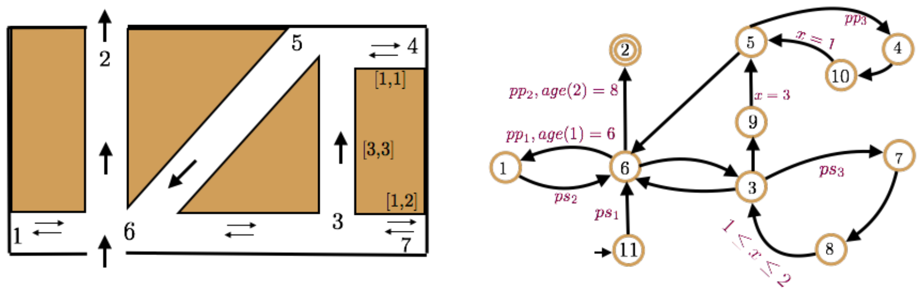


Figure 4 A simple maze. Every junction, dead end, entry point or exit point is called a place, numbered from 1 to 7. 6 is the entry, 2 the exit, 1, 7 and 4 are dead ends. Time intervals denote the time taken between adjacent places; e.g., between 1 and 2 time units must elapse between places 3 and 7. On the right, is the TPDA model of the maze.

are well-nested with respect to the stack. We can also handle clocks which are reset and checked in consecutive transitions.

For the general model (one stack + any number of clocks), we can use optimizations to reduce the number of states of the tree automaton to $(M \times T)^{2|X|+2} \times 2^{2|X|+1}$, where $|X|$ is the number of clocks, M is 1 plus the maximum constant and T is the number of transitions. To see this, consider the worst case scenario, where a state of the tree automaton has $|X|$ hanging points and $|X|$ reset points. In total there can be $2|X| + 2$ active points including the left and right end-points of the non-trivial block. After a combine operation, we can forget a point i of the new state, if it is the case that every clock x reset at the transition (guessed) at point i is also reset at some transition at a point after i. Following this strategy, if we aggressively forget as many points as we can, we will have at most $|X|$ internal (reset) active points between the left and right end-points of the non-trivial block. Thus, we reduce the number of active points from $3|X| + 2$ to $2|X| + 2$.

As a proof of concept, we have implemented our approach with these optimizations. We will now describe some examples we modelled and their experimental results. These experiments were run on a 3.5 GHz i5 PC with 8GB RAM, with number of cores=4.

A Modeling Example : Maze with Constraints

As an interesting example, we model a situation of a robot successfully traversing a maze respecting multiple constraints (see Figure 4). These constraints may include logical constraints: the robot must visit location 1 before exit, or the robot must load something at a certain place i and unload it at place j (so number of visits to i must equal visits to j). We may also have local and global time constraints which check whether adjacent places are visited within a time bound, or the total time taken in the maze is within a given duration. We show below, via an illustrative example, that certain classes of such constraints can be converted into a 1-clock TPDA.

One can go from place p to some of its adjacent place q if there is an arrow from place p to place q. In addition, the following types of constraints must be respected.

1. *Logical constraints* specify certain order between visiting places, the number of times (upper/lower bounds) to visit a place or places, and so on. The logical constraints we have in our example are (a) place 1 must be visited exactly once, (b) from the time we enter the maze, to visiting place 1, one must visit place 7 (load) and place 4 (unload) equal number of times, and at any point of time, the number of visits to place 7 is not less than number of visits to place 4. (c) from visiting place 1 to exiting the maze, one

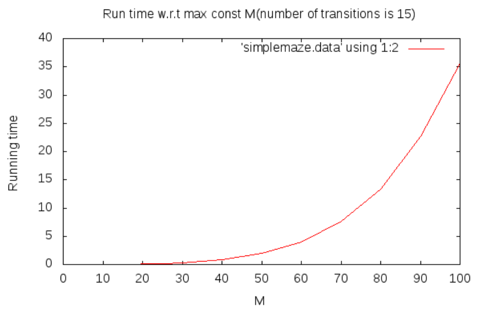
must visit place 7 and place 4 equal number of times and, at any point during time, number of visits to place 7 is not less than number of visits to place 4.

2. *Local time constraints* specify time intervals which must be respected while going from a place to its adjacent place. The time taken from some place i to an adjacent place j is given as a closed interval $[a, b]$ along with the arrow. One cannot spend any time between a pair of adjacent places other than the ones specified in the maze. For example, the time bound for going from place 7 to 3 is given, while the time taken from place 3 to place 7 and place 6 to place 1 is zero ([0,0]), since it is not mentioned. Further, one cannot stay in any place for non-zero duration.
3. *Global time constraints* specify total time that can elapse between visiting two places. From entering the maze to visiting of place 1, time taken should be exactly m units (a parameter). From visiting place 1 to exit, time should be exactly n units (another parameter).

A maze respecting multiple constraints as above is converted into a 1-clock TPDA. While the details of this conversion are given in [5, Appendix E], the main idea is to encode local time bounds with the clock which is reset on all transitions. A logical constraint specifying equal number of visits to places p_1, p_2 is modelled by pushing symbols while at p_1, and popping them at p_2. Likewise, if there is a global time constraint that requires a time elapse in $[a, b]$ between the entry and some place p, then push on the stack at entry, and check its age while at p. Note that these are *well-nested* properties.

To check the existence of a legitimate path in the maze respecting the constraints, our tool checks the existence of a run in the TPDA. By running our tool on the TPDA constructed (and fixing the parameters to be $m = 7$, $n = 8$), we obtain the following run: (described as a sequence of pairs the form : State, Entry time stamp in the state) $(6, 0.0) \to (3, 0.0) \to (7, 0.0) \to (3, 1.0) \to (7, 1.0) \to (3, 2.0) \to (5, 5.0) \to (4, 5.0) \to (5, 6.0) \to (4, 6.0) \to (5, 7.0) \to (6, 7.0) \to (1, 7.0) \to (6, 7.0) \to (3, 7.0) \to (7, 7.0) \to (3, 9.0) \to (7, 9.0) \to (3, 10.0) \to (5, 13.0) \to (4, 13.0) \to (5, 14.0) \to (4, 14.0) \to (5, 15.0) \to (6, 15.0) \to (2, 15.0)$.

The scalability is assessed by instantiating the maze for various choices of maximum constants used, as well as number of transitions. The running times with respect to various choices for the maximum constant are plotted on the right. More examples can be found in [5, Appendix E].



7 Conclusion

We have obtained a new construction for the emptiness checking of TPDA, using tree-width. The earlier approaches [1], [2] which handle dense time and discrete time push down systems respectively use an adaptation of the well-known idea of timed regions. The technique in [2] does not extend to dense time systems, and it is not clear whether the approach in [1] will work for say, multi stack push down automata even with bounded scope/phase restrictions. Unlike this, our approach is uniform : all our proofs except the tree automaton for realizability already work even if we have open guards. Our realizability proof has to be adapted for open guards and this is work under progress; in this paper, we focussed on closed guards to obtain an efficient tool based on our theory. Likewise, our proofs can be extended to bounded phase/scope/rounds multi stack timed push down automata : we need to show a bound on the tree-width, and then adapt the tree automaton construction for the system automaton. The tree automaton checking realizability requires no change. With the theoretical improvements in this paper, we implement our approach and examine its performance on real examples. To the best of our knowledge, this is the first tool implementing timed push down systems. We plan to optimize our implementation to get a more robust and scalable tool. For the subclasses we have, it would be good to have a characterization and automatic translation (currently done by hand) that replaces well-nested clock constraints by stack edges, thus leading to better implementability.

Acknowledgments. The authors thank Vincent Jugé for insightful discussions on the MSO definability of realizability of TC words.

References

1 P. Abdulla, M. F. Atig, and J. Stenman. Dense-timed pushdown automata. In *LICS Proceedings*, pages 35–44, 2012.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Jari Stenman. The minimal cost reachability problem in priced timed pushdown systems. In *Language and Automata Theory and Applications - 6th International Conference, LATA 2012, A Coruña, Spain, March 5-9, 2012. Proceedings*, pages 58–69, 2012.

3 C. Aiswarya and P. Gastin. Reasoning about distributed systems: WYSIWYG (invited talk). In *FSTTCS Proceedings*, pages 11–30, 2014.

4 S. Akshay, P. Gastin, and S. Krishna. Analyzing timed systems using tree automata. In *CONCUR Proceedings*, 2016.

5 S. Akshay, P. Gastin, S. N. Krishna, and I. Sarkar. Towards an efficient tree automata based technique for timed systems. *CoRR*, abs/1707.02297, 2017. URL: `http://arxiv.org/abs/1707.02297`.

6 R. Alur and D. Dill. A theory of timed automata. *In TCS*, 126(2):183–235, 1994.

7 G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 200–236, 2004.

8 A. Bouajjani, R. Echahed, and R. Robbana. On the automatic verification of systems with continuous variables and unbounded discrete data structures. In *Hybrid Systems II*, pages 64–85, 1994.

9 L. Clemente and S. Lasota. Timed pushdown automata revisited. In *LICS Proceedings*, pages 738–749, 2015.

10 B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. CUP, 2012.

11 A. Cyriac, P. Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR Proceedings*, pages 547–561, 2012.

12 Guoqiang Li, Xiaojuan Cai, Mizuhito Ogawa, and Shoji Yuen. Nested timed automata. In *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, pages 168–182, 2013.

13 P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL Proceedings*, pages 283–294, 2011.

14 Ashutosh Trivedi and Dominik Wojtczak. Recursive timed automata. In *ATVA Proceedings*, pages 306–324, 2010.