On Synthesis of Timed Regular Expressions

Ziran Wang
Key Lab. of System Software (CAS)
Institute of Software, Chinese Academy of Sciences,
& University of Chinese Academy of Sciences
Beijing, China
wangzr@ios.ac.cn

Jie An*
National Key Lab. of Space Integrated Information System
Institute of Software, Chinese Academy of Sciences,
& University of Chinese Academy of Sciences
Beijing, China
anjie@iscas.ac.cn

Naijun Zhan*
School of Computer Science & MoE Key Lab.
of High Confidence Software Technologies
Peking University
Beijing, China
njzhan@pku.edu.cn

Miaomiao Zhang
School of Computer Science and Technology
Tongji University
Shanghai, China
miaomiao@tongji.edu.cn

Zhenya Zhang
Kyushu University
Fukuoka, Japan.
zhang@ait.kyushu-u.ac.jp

***Abstract*—Timed regular expressions serve as a formalism for specifying real-time behaviors of Cyber-Physical Systems. In this paper, we consider the synthesis of timed regular expressions, focusing on generating a timed regular expression consistent with a given set of system behaviors including positive and negative examples, i.e., accepting all positive examples and rejecting all negative examples. We first prove the decidability of the synthesis problem through an exploration of simple timed regular expressions. Subsequently, we propose our method of generating a consistent timed regular expression with minimal length, which unfolds in two steps. The first step is to enumerate and prune candidate parametric timed regular expressions. In the second step, we encode the requirement that a candidate generated by the first step is consistent with the given set into a Satisfiability Modulo Theories (SMT) formula, which is consequently solved to determine a solution to parametric time constraints. Finally, we evaluate our approach on benchmarks, including randomly generated behaviors from target timed models and a case study.**

***Index Terms*—Timed regular expressions, Synthesis, Passive learning, Real-time behaviors, Cyber-physical systems**

I. Introduction

In their seminal work [1], Asarin et al. introduced timed regular expressions (TRE) as a formalism for specifying real-time behaviours of timed systems, and demonstrated the equivalent expressiveness of generalized extended TRE with timed automata (TA) [2] proposed by Alur and Dill. Their relationship reflects the classic correspondence between finite-state automata (FSA) and regular expressions (RE), where both formalisms describe the same class of languages but offer complementary advantages. TAs excel at modeling stateful transitions with timing constraints, whereas TREs provide a concise, compositional syntax for specifying timed patterns. This duality not only underscores the theoretical expressiveness of TREs but also motivates their practical utility in learning and synthesis, such as REs have been widely adopted for pattern inference in untimed settings.

* Corresponding authors.

As known, TA is one of the most popular and widely used formal models for real-time systems. For scheduling analysis, as discussed in [3] [4], most of the task models are subclasses of TA (see Figure 4 in [3]). Specifically, a Di-gragh [5] characterized task can be modeled by a labeled time automaton (i.e., a task automaton in [3]), by introducing a clock recording the separation time of two successive jobs.

TREs have also been found numerous applications in modeling and monitoring of real-time systems and cyber-physical systems, similar to how regular expressions play a key role in processing text and system logs. For instance, the timed pattern matching problem [6], [7], [8], [9] involves identifying all subsegments of a signal or system trace that match a given TRE which serves as a timed pattern for real-time specifications. There are also several applications to mining TRE-formed specifications from traces of the QNX Neutrino RTOS and an automobile CAN bus [10], [11].

However, these works, both for scheduling analysis and pattern matching, rely on the assumption that the models are known beforehand. Therefore, for analyzing black-box systems or gray-box systems, where the models are unavailable, a key challenge is to learn a formal model from observable real-time system behaviors. For instance, consider a gray-box task system where we know the type of task model, but the parameters (e.g., the deadline and WCET of jobs) are unknown, and we want to confirm the parameters. The tasks are running on a (or multiple) processor(s) with a certain scheduling strategy, and we can only access the traces of processor behaviors: receiving a job, preemption, completion of a job, and when these behaviors are taken. As shown in [3], a task system can be modeled by a TA. To determine the unknown parameters, we can infer or synthesize a formal model (e.g., a TA or a TRE) from the observed behaviors. Section II provides an illustrative example for the Earliest Deadline First (EDF) scheduling in a gray-box setting.

In recent decades, passive learning involving synthesizing

formal models from system behaviors has garnered significant interest. Prior research has proposed methods to automatically generate regular expressions from examples [12], [13], [14], [15], natural languages [16], [17], [18] and multiple modals [19]. Learning finite-state automata from examples also attracted attention since 1970s [20], [21], [22]. Alternatively, in [23], Angluin proposed an active learning algorithm named L^* within the Minimally Adequate Teacher (MAT) framework, where a learner learns a regular language from a teacher by making membership and equivalence queries. For timed models, both passive and active learning efforts have primarily concentrated on learning various subclasses of timed automata, e.g., event-recording automata [24], [25], real-time automata [26], [27], [28], [29], [30], timed automata with one-clock [31], [32], [33], Mealy machine with one timer [34], [35], deterministic timed automata [36], [37], [38], [39], [40], etc. Recently, an inference algorithm for timed partial orders (TPO) was presented in [41]. TPO can also be viewed as a subset of timed automata.

However, the subclasses of timed automata mentioned above are deterministic. Therefore, existing learning or synthesizing algorithms for these models cannot be directly applied to the synthesis of TRE involving nondeterministic behaviors. The reduction from TA to TRE (i.e., without renaming function) may not always be possible. Moreover, for deterministic cases, synthesizing TA is irrelevant to minimizing TREs. The latter itself is a difficult problem, even for traditional regular expressions, it is still PSPACE-complete [42].

Contributions. In this paper, we consider the synthesis of TRE from system behaviors, highlighting the following several distinctions from the extensively studied synthesis problem of traditional regular expressions (RE).

- We establish the decidability of synthesizing a TRE consistent with a given set of timed words, which includes positive and negative examples, by exploring a notion of simple timed regular expressions and their falsifiability.
- Subsequently, we propose a method for a particular synthesis problem known as minimal synthesis, aiming to generate a consistent TRE with minimal length. The synthesis process begins by generating a template and then determining the timing constraints. Specifically, in the first step, we generate an untimed version of TRE, referred to as parametric TRE, serving as the template. To achieve a TRE with minimal length, we systematically enumerate such templates starting from a length k.
- Additionally, we introduce two strategies for pruning unnecessary candidates to expedite the enumeration. In the second step, we encode the requirement that a candidate parametric TRE remains consistent with the given set into a formula in Satisfiability Modulo Theories (SMT) [43], which is then solved to determine a solution to the parametric timing constraints within the parametric TRE. If no solution is found, the process jumps to the first step. We provide the proof of termination and correctness of the synthesis process at the end.

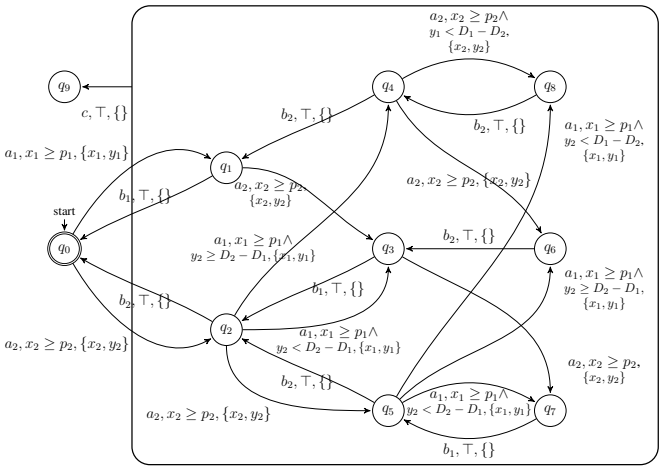

Fig. 1. The timed automaton of the scheduling example.

- Finally, we implement and evaluate our synthesis process on benchmarks, including randomly generated behaviors of target timed models and a case study.

Outline. Section II provides a motivating example, and then Section III recalls some basic notions of timed regular expressions. The TRE synthesis problem ia formalized in Section IV, and then the decidability is shown in Section V. The two-step synthesis process is presented in Section VI and evaluated in Section VII. Finally, Section VIII concludes the paper.

The omitted proofs of the lemmas and theorems are available in the full version [44].

II. Illustrative Example

To illustrate a potential motivation for synthesizing formal models from the execution traces of real-time systems, we present the following application scenario: Consider a task model where the minimum interarrival time and deadline are partly unknown and are determined by external factors. However, the execution time is controllable and can be influenced by the resources allocated to the task. For instance, if the tasks are requests from a website, the execution times will depend on the server resources allocated by the service provider. Before commencing the formal operation of the system, it is crucial to determine the execution time required to ensure the schedulability of the system. To achieve this, trial runs are conducted to analyze the parameters of the minimum interarrival time and deadline from the execution traces.

For instance, let us consider a case of a preemptive EDF scheduling problem with 2 tasks as follows. Two sporadic tasks τ_1, τ_2 run in parallel on one processor, and they have unknown minimum interarrival times p_i, deadlines D_i and controllable worst case execution time C_i, where $i = 1, 2$. At the same time, some constraints of the parameters are known: $p_1 \leq p_2$, $D_1 = p_1$, and $p_2 < D_2 \leq 2p_2$. Based on the known information, the system of task model with the preemptive EDF strategy can be modeled as a TA shown in Fig. 1. With preemptive EDF scheduling, the job currently executing on the processor is the first one in the priority queue determined by

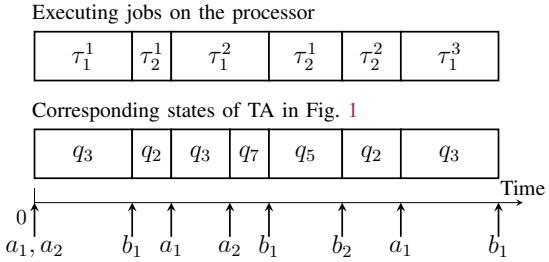

Fig. 2. A possible schedule scenario over time with the executing jobs on the processor and the corresponding states of TA in Fig. 1. τ_i^j denotes the j-th job of the task τ_i.

deadlines. Since $D_1 = p_1$, there is at most 1 job of τ_1 before a deadline is missed. Likewise, there are at most 2 jobs of τ_2 in the queue before deadline misses. In Fig. 1, state q_0 stands for system idling, q_1 (resp. q_2) for only τ_1 (resp. τ_2) in the queue, q_3 for τ_1, τ_2 in the queue, q_4 for τ_2, τ_1 in the queue, q_5 for τ_2, τ_2 in the queue, q_6 for τ_2, τ_1, τ_2 in the queue, q_7 for τ_1, τ_2, τ_2 in the queue, and q_8 for τ_2, τ_2, τ_1 in the queue. And q_9 is the state for missing deadlines. Event a_i stands for the arrival of jobs of τ_i, and b_i for the completion of τ_i, where $i = 1, 2$. Event c stands for the claim of errors when any deadline is missed. Clock x_i is used to record the separation time of jobs of τ_i, and clock y_i is used to record the time from the latest jobs of τ_i released up to now, where $i = 1, 2$. A transition of the TA can be taken when the event happens and the value of the clocks satisfies the inequalities (guards), and after taking the transition, some clocks are reset to 0. For instance, the transition from q_1 to q_3 means that the next job of τ_2 arrives and the current job of τ_1 is not yet completed ($x_2 \geq p_2$), then the clocks x_2, y_2 for τ_2 are reset to 0, and the system moves to q_3. And for any state except q_0, q_9, the transition to q_9 is taken if a deadline miss is claimed.

Fig. 2 illustrates a possible schedule scenario over time with the executing jobs and corresponding states of the TA in Fig. 1. τ_i^j denotes the j-th job of the task τ_i. At the beginning, suppose each task has a job in the queue. Since the priority of τ_1 is higher than that of τ_2 (as $D_1 < D_2$), the TA is at State q_3 and the processor executes τ_1^1 until finishing it. After that, the job τ_2^1 is executed for a while, then the second job τ_1^2 of the task τ_1 comes and preempts the execution of τ_2^1. When the second job of τ_2 comes, the TA jumps from q_3 to q_7. After finishing the second job of τ_1, the TA jumps from q_7 to q_5 and the processor completes the execution of τ_2^1. After that, the processor executes the second job of τ_2 for a while, before the third job of τ_1 comes to preempt its execution.

To determine the parameters, we can learn a formal model, say a TA or a TRE, from the observed behaviors. After that, we can determine the condition for the execution times C_1 and C_2 so that the system scheduling is feasible. We present it as a case study in Section VII-C.

III. Preliminaries

In this section, we review the basic notions and results on timed regular expressions. Please refer to [1] for a comprehensive introduction.

Let $\mathbb{R}_{\geq 0}$ and $\mathbb{N}$ represent non-negative reals and natural numbers, respectively. *An integer-bounded interval* I is either $[l, u]$, $(l, u]$, $[l, u)$, (l, u), $[l, \infty)$ or (l, ∞), where $l, u \in \mathbb{N}$ such that $l \leq u$. Let Σ be a fixed finite alphabet. In practice, an event $\sigma \in \Sigma$ denotes an action of a system under consideration.

A *timed word* is a finite sequence $\omega = (\sigma_1, t_1)(\sigma_2, t_2) \cdots (\sigma_n, t_n) \in (\Sigma \times \mathbb{R}_{\geq 0})^*$, where t_i represents the delay time before taking action σ_i for all $1 \leq i \leq n$. Particularly, the special symbol ε represents the *empty word*. For a timed word ω, we define two functions $\lambda : (\Sigma \times \mathbb{R}_{\geq 0})^* \to \mathbb{R}_{\geq 0}^*$ and $\mu : (\Sigma \times \mathbb{R}_{\geq 0})^* \to \Sigma^*$ to get the sequence of delay times (i.e., $\lambda(\omega) = t_1, t_2, \ldots, t_n \in \mathbb{R}_{\geq 0}^*$) and the sequence of events (i.e., $\mu(\omega) = \sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma^*$), respectively. For instance, let $\omega = (a, 0.2)(b, 0)(a, 6)$, then we have $\lambda(\omega) = 0.2, 0, 6$ and $\mu(\omega) = aba$. A *timed language* is a set of timed words.

According to [1], the full set of timed regular expressions, named generalized extended timed regular expressions ($\mathcal{GEE}(\Sigma)$), admits the following operations: *concatenation* ($\cdot$), *union* ($\vee$), *time restriction* ($\langle\,\rangle_I$), *Kleene star* (*), *absorbing concatenation* ($\circ$), *absorbing iteration* ($^{\circledast}$), *intersection* ($\wedge$), and *renaming* (θ). It is proved that $\mathcal{GEE}(\Sigma)$ has the same expressive power as timed automata [2]. Renaming function is needed to show the expressiveness of $\mathcal{GEE}(\Sigma)$ but not easy to infer from system behaviors. In this paper, we consider the standard timed regular expressions ($\cdot$, *, $\vee$ and $\langle\,\rangle_I$) and the extended timed regular expressions ($\circ$ and $^{\circledast}$ additionally). Since the extended timed regular expressions ($\mathcal{EE}(\Sigma)$) have the same expressive power as standard timed regular expressions ($\mathcal{E}(\Sigma)$), we thus use the following definition of standard timed regular expressions directly.

Definition 1 (Timed regular expressions). The syntax of timed regular expressions (TRE) over Σ is as follows:

$$\varphi := \varepsilon \mid \underline{\sigma} \mid \langle\varphi\rangle_I \mid \varphi \cdot \varphi \mid \varphi \vee \varphi \mid \varphi^*$$

where $\underline{\sigma}$ for every event $\sigma \in \Sigma$ and I is an integer-bounded interval, $\cdot$, $\vee$, and $*$ stand for concatenation, disjunction, and Kleene star, respectively. The semantics of TRE $[\![\ ]\!] : \mathcal{E}(\Sigma) \to 2^{(\Sigma \times \mathbb{R}_{\geq 0})^*}$, which associates a TRE with a set of timed words, is inductively defined as follows:

$$[\![\varepsilon]\!] = \{\varepsilon\}, \quad [\![\underline{\sigma}]\!] = \{(\sigma, t) : t \in \mathbb{R}_{\geq 0}\}, \quad [\![\langle\varphi\rangle_I]\!] = [\![\varphi]\!] \cap \{\omega : \sum \lambda(\omega) \in I\}, \quad [\![\varphi_1 \cdot \varphi_2]\!] = [\![\varphi_1]\!] \cdot [\![\varphi_2]\!],$$
$$[\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!], \quad [\![\varphi^*]\!] = \bigcup_{i=0}^{\infty} ([\![\underbrace{\varphi \cdot \ldots \cdot \varphi}_{i \text{ times}}]\!])$$

A TRE represents a timed language. Two timed words ω_1 and ω_2 can be *distinguished* by a TRE φ if ($\omega_1 \in [\![\varphi]\!]$ *and* $\omega_2 \notin [\![\varphi]\!]$) or ($\omega_1 \notin [\![\varphi]\!]$ *and* $\omega_2 \in [\![\varphi]\!]$). To determine whether $\omega \in [\![\varphi]\!]$ is called as the *membership decision*, which can be done by a standard method [1]. The meaning of $\underline{\sigma}$ for every event $\sigma \in \Sigma$ represents the occurrence of σ after delaying an arbitrary time. The *time restriction (constraint)* $\langle\varphi\rangle_I$ limits the total time passages in the interval I over timed words ω in $[\![\varphi]\!]$. It is obvious that $[\![\langle\varepsilon\rangle_I]\!] = \{\varepsilon\}$. Other operations $\vee$, $\cdot$ and

* satisfy the standard Kleene algebra and we use the following conventions $\sigma = \langle \underline{\sigma} \rangle_{[0,0]}$, $\varphi^0 = \varepsilon$, $\varphi^+ = \varphi \cdot \varphi^*$, and $\varphi^{i+1} = \varphi^i \cdot \varphi = \varphi \cdot \varphi^i$. We will omit the concatenation symbol $(\cdot)$ if the context is clear. Therefore, compared to regular expressions, the most important new feature here is the time restriction. The following examples further illustrate it.

$$\varphi_1 = \langle \underline{\sigma} \rangle_{[0,0]}$$
$$[\![\varphi_1]\!] = \{(\sigma, 0)\}$$
$$\varphi_2 = \langle \underline{\sigma} \rangle_{[3,7]}$$
$$[\![\varphi_2]\!] = \{(\sigma, t) : t \in [3,7]\}$$
$$\varphi_3 = \langle \underline{\sigma_1} \rangle_{[1,3]} \cdot \langle \underline{\sigma_2} \rangle_{[2,4]}$$
$$[\![\varphi_3]\!] = \{(\sigma_1, t_1)(\sigma_2, t_2) : t_1 \in [1,3] \wedge t_2 \in [2,4]\}$$
$$\varphi_4 = \langle \langle \underline{\sigma_1} \rangle_{[1,3]} \cdot \underline{\sigma_2} \rangle_{[2,4]}$$
$$[\![\varphi_4]\!] = \{(\sigma_1, t_1)(\sigma_2, t_2) : t_1 \in [1,3] \wedge t_1 + t_2 \in [2,4]\}$$
$$\varphi_5 = \langle \langle \underline{\sigma_1} \rangle_{[1,3]} \cdot \langle \underline{\sigma_2} \rangle_{[2,4]} \rangle_{[5,6]}$$
$$[\![\varphi_5]\!] = \{(\sigma_1, t_1)(\sigma_2, t_2) : t_1 \in [1,3] \wedge t_2 \in [2,4] \wedge t_1 + t_2 \in [5,6]\}$$
$$\varphi_6 = \langle \underline{\sigma}^* \rangle_{[3,7]}$$
$$[\![\varphi_6]\!] = \left\{(\sigma, t_1)(\sigma, t_2) \cdots (\sigma, t_n) : n \in \mathbb{N} \wedge \sum\nolimits_{i=1}^{n} t_i \in [3,7]\right\}$$
$$\varphi_7 = \left(\langle \underline{\sigma} \rangle_{[3,7]}\right)^*$$
$$[\![\varphi_7]\!] = \left\{(\sigma, t_1)(\sigma, t_2) \cdots (\sigma, t_n) : n \in \mathbb{N} \wedge \bigwedge\nolimits_{i=1}^{n} t_i \in [3,7]\right\}$$

We reuse the function μ to get the corresponding regular expression (RE) $\mu(\varphi)$ by erasing the time restrictions in a TRE φ. For instances, $\mu(\varphi_1) = \mu(\varphi_2) = \sigma$, $\mu(\varphi_3) = \mu(\varphi_4) = \mu(\varphi_5) = \sigma_1 \cdot \sigma_2$, and $\mu(\varphi_6) = \mu(\varphi_7) = \sigma^*$.

A *TRE syntax tree* (see the example in Fig. 3 in Section VI) allows to represent a TRE with a tree structure, where internal nodes are labeled by operations and leaf nodes are labeled by actions in Σ. An internal node labeled with a unary operation (*) has exactly one child. An internal node labeled with a binary operation ($\cdot$ or $\vee$) has exactly two children. Each node is equipped with a time restriction $\langle\rangle_I$ as an inherent property.

IV. Problem Formulation

In this section, we first define the TRE synthesis problems of our interest. Then we discuss several distinctions between synthesizing TRE and traditional RE.

A. Problem Statements

In general, our purpose is to synthesize a TRE φ from a given finite set of timed words $\Omega \subseteq (\Sigma \times \mathbb{R}_{\geq 0})^*$. One particular setting is that the set of examples is given by a pair $\Omega = (\Omega_+, \Omega_-)$, where Ω_+ contains the positive examples and Ω_- contains the negative ones.

Problem 1 (Synthesis problem). Given a finite set of timed words $\Omega = (\Omega_+, \Omega_-)$, to synthesize a TRE φ such that for all $\omega \in \Omega_+, \omega \in [\![\varphi]\!]$ *and* for all $\omega \in \Omega_-, \omega \notin [\![\varphi]\!]$, or to claim that there does not exist such TRE. In the first case, we say ϕ recognizes Ω.

To specify ideal expressions, we can introduce measures in TRE. The motivation is that for Problem 1 we need some measure to determine an interesting expression since there may be infinitely many such expressions that recognize a given set Ω. Formally, a measure $\mathcal{M} : \mathcal{E}(\Sigma) \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ assigns a real value to every expression so that we can compare them.

However, there does not appear to be any uniform agreement on how to measure the size of even untimed regular expressions on the alphabet Σ. An obvious measure is the ordinary length, i.e., the total number of symbols including parentheses. Another measure is the formula length based on the reverse Polish form. Evidently, reverse polish length is the same as the number of nodes in the syntax tree for the expression.

For TRE, we are interested in the number of nodes in the syntax tree with considering time restriction $\langle\rangle_I$ as a property for each node. We say the *length* of a TRE is the number of nodes in the corresponding syntax tree.

Problem 2 (Minimal synthesis problem). Given a finite set of timed words Ω, synthesize a TRE φ with minimal length recognizing Ω, or claim that there is no such TRE.

B. Differences from synthesizing regular expressions

Synthesizing time restrictions. We not only synthesize untimed expressions but also time constraints. Time constraints are not only for the letters in expressions but also for the constraints over the operations (subformulas). TRE containing time constraints only for letters is equivalent to real-time automata, which is a kind of timed automata with only one clock and resets it at every transition.

Rejecting negative examples. An RE rejects a string over Σ iff the string cannot match the expression. However, in our case, a TRE rejects a timed word over $\Sigma \times \mathbb{R}_{\geq 0}$ is additionally due to timing information. Therefore, given a negative example ω and a TRE φ, the string $\mu(\omega)$ may match the corresponding untimed RE. It brings the obstacle of utilizing the current pruning methods of synthesizing regular expressions to the synthesis of TRE.

Pruning structure. Given two RE $r_1 = (\underline{a} \cdot \underline{b})^*$ and $r_2 = (\underline{a} \cdot \underline{b})^* \cdot (\underline{a} \cdot \underline{b})^*$, we have that $r_1 = r_2$ and r_1 is smaller than r_2 in length. If r_1 does not recognize the examples, then we do not need to take r_2 into consideration anymore and thus skip it in the searching process. However, this is not the case for synthesizing TRE. For instance, $\varphi_1 = (\langle \underline{a} \cdot \underline{b} \rangle_{[1,3]})^*$ and $\varphi_2 = (\langle \underline{a} \cdot \underline{b} \rangle_{[1,3]})^* \cdot (\langle \underline{a} \cdot \underline{b} \rangle_{[4,6]})^*$, we have $[\![\varphi_1]\!] \subset [\![\varphi_2]\!]$. Even if we cannot synthesize ideal intervals on the untimed expression $(ab)^*$, we cannot skip the equivalent expression $(ab)^* \cdot (ab)^*$.

Equivalent structure. For example, the RE structure $(\underline{a} \cdot \underline{b}) \cdot \underline{c}$ is equivalent to $\underline{a} \cdot (\underline{b} \cdot \underline{c})$. But they are not equivalent anymore as we equip every concatenation $\cdot$ with a time restriction, since the time restrictions of the first and the second concatenation constraint the delay times of timed words differently in the two structures. So we cannot skip one of the structures if we have already checked the other.

Limitation. For synthesizing REs from positive and negative examples, if we assume that the two sets are finite and disjoint, then we can always get a valid expression of

which the language is not empty. However, for the case of synthesizing TREs, even if the two sets are finite and disjoint, we may claim that there is no such TRE recognizing the examples. One reason is that we can not generate valid integer-bounded intervals. Another reason is that we need to use other operations, e.g., intersection, if we want to represent some timed languages by TREs. For example, a $\mathcal{GEE}$ expression $(\langle a \cdot b\rangle_{[1,1]} \cdot c) \wedge (a \cdot \langle b \cdot c\rangle_{[1,1]})$ denoting the set $\{(a,t_1)(b,t_2)(c,t_3) : (t_1+t_2=1) \wedge (t_2+t_3=1)\}$ can not be expressed without intersection.

V. Decidability of TRE Synthesis Problems

For Problem 1, one can imagine that there exists infinite many of TREs accepting all positive examples, for instance, since Σ is finite, $\varphi = \langle\langle\underline{\sigma_1}\rangle_{[0,\infty)} \vee_{[0,\infty)} \langle\underline{\sigma_2}\rangle_{[0,\infty)} \vee_{[0,\infty)} \cdots \vee_{[0,\infty)} \langle\underline{\sigma_n}\rangle_{[0,\infty)}\rangle^*\rangle_{[0,\infty)}$, where $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_n\}$ is a "naïve" solution TRE accepting all timed words $w \in (\Sigma \times \mathbb{R}_{\geq 0})^*$. However, when considering negative examples, the learnt TRE should reject all negative examples because of at least one of the following two reasons: untimed sequences are unmatched or the delay time information is not acceptable. Considering the second condition, given a positive example ω and a negative example ω' such that $\mu(\omega) = \mu(\omega')$, the time restrictions in a candidate TRE should be able to reject $\lambda(\omega')$. Therefore, we introduce the *simple timed regular expression* to decide whether a TRE is able to distinguish any pair of positive and negative examples.

Definition 2 (Simple elementary language [39]). Given a timed word $\omega = (\sigma_1,t_1)(\sigma_2,t_2)\cdots(\sigma_n,t_n)$, the corresponding *simple elementary language* (sEL) is a timed language $\mathcal{SE}(\omega) = \{(\sigma_1,t'_1)(\sigma_2,t'_2)\cdots(\sigma_n,t'_n) : t'_1,t'_2,\cdots,t'_n \models \Lambda(\omega)\}$, where $\Lambda(\omega) = \bigwedge \Theta(\omega)$ and $\lambda(\omega)$ satisfies $\Lambda(\omega)$, i.e., $\lambda(\omega) \models \Lambda(\omega)$. $\Theta(\omega)$ is the finite set of tightest integer-bounded time constraints for the sums of all possible consecutive sequences of delay times of w. i.e. $\forall 1 \leq j \leq k \leq n$, let d be the integer part of $\sum_{i=j}^{k} t_i$ and e be the fractional part, if $e > 0$ then inequality $d < \sum_{i=j}^{k} t_i < d+1$ is in $\Theta(\omega)$, otherwise equality $\sum_{i=j}^{k} t_i = d$ is in $\Theta(\omega)$.

Example 1. Considering timed words $\omega_1 = (a,1.2)(a,2.2)$, $\omega_2 = (a,1.2)(a,2.6)$, $\omega_3 = (a,1.5)(a,2.6)$, we have $\mathcal{SE}(\omega_1) = \mathcal{SE}(\omega_2) = [\![\langle\langle\underline{a}\rangle_{(1,2)} \cdot \langle\underline{a}\rangle_{(2,3)}\rangle_{(3,4)}]\!]$, and $\mathcal{SE}(\omega_3) = [\![\langle\langle\underline{a}\rangle_{(1,2)} \cdot \langle\underline{a}\rangle_{(2,3)}\rangle_{(4,5)}]\!]$

Lemma 1. Given two timed words ω_1 and ω_2, if $\mathcal{SE}(\omega_1) = \mathcal{SE}(\omega_2)$, then ω_1 and ω_2 cannot be distinguished by any TRE.

However, not all elementary languages necessarily have a corresponding TRE whose semantics are equal to the elementary language. For instance, given $\omega_1 = (a,1.4)(a,1.4)(a,1.4)$ we have $\mathcal{SE}(\omega_1) = \{(a,\tau_1)(a,\tau_2)(a,\tau_3) : \tau_1,\tau_2,\tau_3 \in (1,2), \tau_1+\tau_2, \tau_2+\tau_3 \in (2,3), \tau_1+\tau_2+\tau_3 \in (4,5)\}$. There does not exist a TRE φ s.t. $[\![\varphi]\!] = \mathcal{SE}(\omega_1)$, since τ_2 appears in both constraints of $\tau_1+\tau_2$ and $\tau_2+\tau_3$, which is not allowed according to Definition 1. Hence, we loose the requirement of sEL to define simple timed regular expressions as follows.

Definition 3 (Simple timed regular expressions). A simple timed regular expression (sTRE) φ contains only two operations $\cdot$ and $\langle\rangle_I$, where all time restrictions I are in the form of $(d,d+1)$ or $[d,d]$, and $d \in \mathbb{N}$.

Given a timed word ω, we denote $\Phi(\omega)$ the finite set of its corresponding sTRE, and write φ_ω for one of the corresponding sTRE, i.e., $\varphi_\omega \in \Phi(\omega)$.

Lemma 2. Given a timed word ω, the number of its corresponding sTRE φ (i.e., $\omega \in [\![\varphi]\!]$) is at most $2^{(n^2+n)/2}$, where $n = |\omega|$.

The following lemma shows that the simple elementary language of a timed word is identical to the language intersection of all its corresponding sTRE.

Lemma 3. Given a timed word ω, $\mathcal{SE}(\omega) = \bigcap_{\varphi_\omega \in \Phi(\omega)} [\![\varphi_\omega]\!]$.

Definition 4 (Obscuration). A timed word ω is obscured by a set of timed words Ω, iff $\forall \varphi_\omega \in \Phi(\omega), \exists \omega' \in \Omega, \omega' \in [\![\varphi_\omega]\!]$. A timed word ω is not obscured by a set of timed words Ω, iff $\exists \varphi_\omega \in \Phi(\omega), \forall \omega' \in \Omega, \omega' \notin [\![\varphi_\omega]\!]$.

Example 2. Consider $\omega_1 = (a,1.5)(a,2.6)(a,1.5)$, $\omega_2 = (a,1.2)(a,2.6)(a,1.5)$, $\omega_3 = (a,1.5)(a,2.6)(a,1.2)$. Timed word ω_1 is not obscured by $\{\omega_2\}$ or $\{\omega_3\}$, but is obscured by $\{\omega_2,\omega_3\}$. For instance, $\omega_1,\omega_3 \in [\![\langle\langle\underline{a}\cdot\underline{a}\rangle_{(4,5)}\cdot\underline{a}\rangle_{(5,6)}]\!]$, $\omega_2 \notin [\![\langle\langle\underline{a}\cdot\underline{a}\rangle_{(4,5)}\cdot\underline{a}\rangle_{(5,6)}]\!]$, $\omega_1,\omega_2 \in [\![\langle\underline{a}\cdot\langle\underline{a}\cdot\underline{a}\rangle_{(4,5)}\rangle_{(5,6)}]\!]$, $\omega_3 \notin [\![\langle\underline{a}\cdot\langle\underline{a}\cdot\underline{a}\rangle_{(4,5)}\rangle_{(5,6)}]\!]$.

Lemma 4. Given a timed word ω, the obscuration of it w.r.t a set of timed words Ω is decidable in at most $2^{(n^2+n)/2}m$ TRE membership decisions, where $n = |\omega|$ and $m = |\Omega|$.

Lemma 5. If a positive example $\omega \in \Omega_+$ is obscured by the set of negative examples Ω_-, for all TRE ϕ such that $\omega \in [\![\phi]\!]$, there exists $\omega' \in \Omega_-$ such that $\omega' \in [\![\phi]\!]$.

Theorem 1. Problem 1 has a solution TRE w.r.t a given sets $\Omega = (\Omega_+,\Omega_-)$ iff $\forall \omega_+ \in \Omega_+$, ω_+ is not obscured by Ω_-. If so, one solution to Problem 1 is $\varphi = \bigvee_{\omega_+ \in \Omega_+} \varphi_{\omega_+}$.

By Lemma 4 and Theorem 1, we have the conclusion that Problem 1 is decidable. In what follows, suppose that Problem 1 has a solution w.r.t a given set Ω, we investigate Problem 2, i.e., synthesizing a TRE with minimal length.

VI. TRE Synthesis

A. Overview of synthesis of TRE with minimal length

For the minimal synthesis problem (Problem 2), the basic idea is that we enumerate TRE with increasing length and check if the current candidate recognizes the given set Ω. However, unlike traditional RE, even for a finite alphabet, the number of k-length TRE is infinite, making it impossible to enumerate TRE in this naïve way.

Therefore, we introduce a structure, named *parametric timed regular expression* (pTRE), equipped with *parametric intervals* replacing of time restrictions, which is suitable for enumeration.

Algorithm 1: Synthesizing a TRE with minimal length

input : A sample set $\Omega = (\Omega_+, \Omega_-)$.
output: A minimal TRE φ recognizing Ω.

```
1 k ← 1;
2 while ⊤ do
3     foreach φ+ ← enumerate(k, Ω+) do
4         ξ ← encode(φ+, Ω−) ;          // Encode
           k-length pTRE+ φ+ once get it.
5         flag, φ ← SMTsolver(ξ, φ+);
6         if flag = ⊤ then
7             return φ;
8     k ← k + 1;
```

Algorithm 2: Trivial Enumeration of $p\text{TRE}_+$

input : A required length $k \in \mathbb{N}$; the positive samples Ω_+
output: The set of k-length $p\text{TRE}_+$ S_+.

```
1  rules ← {2, 3, 4} ;
2  S_k ← substitute(rules,♡,k);
3  foreach φ ∈ S_k do
4      rules ← {1};
5      S'_k ← substitute(rules,φ,k);
6      foreach φ' ∈ S'_k do
7          flag ← check_acceptable(φ',Ω+);
8          if flag = ⊤ then
9              S+.add(φ');
10 return S+;
```

Definition 5 (Parametric interval). A parametric interval $\mathbb{I}$ is a 4-tuple (lo, l, u, uo) where two integer variables l, u denote the lower and upper endpoints of the interval, and two Boolean variables lo, uo indicate whether the interval includes the start and end point, respectively.

Definition 6 (Parametric timed regular expressions). The syntax of parametric timed regular expressions (pTRE) over Σ is given by the grammar:

$$\phi := \heartsuit \mid \varepsilon \mid \langle \underline{\sigma} \rangle_{\mathbb{I}} \mid \langle \phi \cdot \phi \rangle_{\mathbb{I}} \mid \langle \phi \vee \phi \rangle_{\mathbb{I}} \mid \langle \phi^* \rangle_{\mathbb{I}}$$

where $\heartsuit$ is a placeholder for further enumeration of pTRE that can be substituted by expression ϕ s.t. $\phi \neq \heartsuit$.

Particularly, a pTRE is termed a *closed* pTRE if it contains no $\heartsuit$. A pTRE is termed an *edge* pTRE if it contains only one $\heartsuit$.

If all parametric intervals $\mathbb{I}$ in a closed pTRE ϕ are determined, we say the resulting TRE φ is an instance of ϕ, denoted as $\varphi \in [\phi]$. Therefore, a pTRE ϕ has infinitely many instances. The *length* of pTRE is defined as the number of nodes in the syntax tree.

We denote a specific instance of a closed pTRE ϕ as $max(\phi)$, where all parametric intervals in ϕ are set to $[0, \infty)$. A time word ω is *accepted* by a closed pTRE ϕ iff ω is accepted by $max(\phi)$.

Lemma 6. The number of pTRE with length $k \in \mathbb{N}$ is finite.

Given a set of positive examples Ω_+, we write $p\text{TRE}_+$ for a closed pTRE accepting all positive examples $\omega_+ \in \Omega_+$. By Lemma 6, the number of $p\text{TRE}_+$ with k length is also finite.

Now, instead of enumerating TRE, we can enumerate $p\text{TRE}_+$. For each $p\text{TRE}_+$ ϕ_+, since it already accepts all positive examples, we encode the requirement that ϕ_+ rejects all negative examples Ω_- into an SMT formula. If a solution to the formula exists, we obtain instances for the parametric intervals inside and derive a TRE φ recognizing Ω. Since we systematically enumerate $p\text{TRE}_+$ with increasing length, it is ensured that the resulting TRE is a minimal one. Algorithm 1 summarizes the synthesis process in a nutshell. Note that there may be a lot of $p\text{TRE}_+$ with a length k, we encode each $p\text{TRE}_+$ with Ω_- into an SMT formula ξ greedily once we get one (Line 3 and Line 4) instead of after collecting all of k-length $p\text{TRE}_+$. If there is a solution to ξ (Line 6), we get a minimal TRE φ recognizing Ω. If there is no solution with k length, we start to find $p\text{TRE}_+$ with length $k+1$ (Line 8).

Before proving the termination and correctness of our method, we present the enumerating details and several pruning techniques utilized (Section VI-B), and then show the details of the encoding (Section VI-C).

B. Enumerating and pruning pTRE

The following lemma claims that searching for minimal TRE (and thus the enumeration of pTRE) does not need to consider ε, since it is obvious that a pTRE with ε has the same semantics as a shorter pTRE without ε.

Lemma 7. ε never appears in a minimal TRE w.r.t a nonempty set Ω.

Therefore, starting with a placeholder $\heartsuit$, we consider the following substitution rule in the enumeration.

1. $\dfrac{}{\heartsuit \to \langle \underline{\sigma} \rangle_{\mathbb{I}}}, \sigma \in \Sigma$ 2. $\dfrac{}{\heartsuit \to \langle \heartsuit^* \rangle_{\mathbb{I}}}$
3. $\dfrac{}{\heartsuit \to \langle \heartsuit \cdot \heartsuit \rangle_{\mathbb{I}}}$ 4. $\dfrac{}{\heartsuit \to \langle \heartsuit \vee \heartsuit \rangle_{\mathbb{I}}}$

Trivial enumeration. The basic idea is that we enumerate pTRE with increasing length from $k = 1$ and check whether the current pTRE is a $p\text{TRE}_+$. More precisely, we start from a placeholder $\heartsuit$ and only utilize the last three rules to generate all pTRE with a given length k. Therefore, at this stage, each pTRE contains no concrete actions. We write S_k for the set of such pTRE. For each k-length pTRE in S_k, we utilize the first rule to substitute all placeholders and thus get a finite set of closed pTRE S'_k. Subsequently, we can select out a k-length $p\text{TRE}_+$ by checking whether a closed pTRE accepts all positive examples. Algorithm 2 summarizes trivial enumeration for $p\text{TRE}_+$ with length $k \in \mathbb{N}$. We have the following conclusion.

Example 3. Suppose the alphabet $\Sigma = \{a, b\}$. For $k = 2$, we first use Rule 2 $\heartsuit \to \langle \heartsuit^* \rangle_{\mathbb{I}}$ to get $S_2 = \{\langle \heartsuit^* \rangle_{\mathbb{I}}\}$. For $k = 3$,

Algorithm 3: Recursive Enumeration with Edge Pruning

input : A required length k; the positive samples Ω_+
output: The set of k-length $p\text{TRE}_+$ S_+.

$S_0 \leftarrow \heartsuit$; $step \leftarrow 0$;
while $step < k$ **do**
  foreach $\phi \in S_0$ **do**
    if ϕ *is not closed* **then**
      S_1.add(ϕ's direct children);
  $step \leftarrow step + 1$;
  foreach $\phi \in S_1$ **do**
    if ϕ *is an edge pTRE* **then**
      edge_pruning(ϕ);
  $S_0 \leftarrow S_1$; $S_1 \leftarrow \emptyset$;
foreach $\phi \in S_0$ **do**
  if ϕ *is closed* **then**
    flag $\leftarrow$ check_acceptable(ϕ,Ω_+);
    if *flag* $= \top$ **then**
      S_+.add(ϕ);
return S_+;

we utilize the last three rules to get $S_3 = \{\langle\langle\heartsuit^*\rangle^*_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\heartsuit \cdot \heartsuit\rangle_{\mathbb{I}}, \langle\heartsuit \vee \heartsuit\rangle_{\mathbb{I}}\}$. Then we utilize the first rule to get $S'_2 = \{\langle\langle\underline{a}\rangle^*_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{b}\rangle^*_{\mathbb{I}}\rangle_{\mathbb{I}}\}$ and $S_3 = \{\langle\langle\langle\underline{a}\rangle^*_{\mathbb{I}}\rangle^*_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\langle\underline{b}\rangle^*_{\mathbb{I}}\rangle^*_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{a}\rangle_{\mathbb{I}} \cdot \langle\underline{a}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{a}\rangle_{\mathbb{I}} \cdot \langle\underline{b}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{b}\rangle_{\mathbb{I}} \cdot \langle\underline{a}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{b}\rangle_{\mathbb{I}} \cdot \langle\underline{b}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{a}\rangle_{\mathbb{I}} \vee \langle\underline{a}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{a}\rangle_{\mathbb{I}} \vee \langle\underline{b}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{b}\rangle_{\mathbb{I}} \vee \langle\underline{a}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}, \langle\langle\underline{b}\rangle_{\mathbb{I}} \vee \langle\underline{b}\rangle_{\mathbb{I}}\rangle_{\mathbb{I}}\}$.

Lemma 8. All $p\text{TRE}_+$ with length $k \in \mathbb{N}$ w.r.t a given positive examples Ω_+ can be generated by Algorithm 2.

Recursive Generation with Edge Pruning. The intuition of trivial enumeration is that all pTREs are generated and checked one-by-one to be $p\text{TRE}_+$. Here we propose a method based on recursive generation and edge pruning. Instead of collecting all pTRE with length k first, we recursively generate a closed pTRE with length k based on all substitution rules, and prune the edge pTRE which is impossible to expand to $p\text{TRE}_+$.

Here we call pTRE ϕ_1 to be one of pTRE ϕ_2's (direct) children iff ϕ_1 can be generated from ϕ_2 by arbitrarily applying generating rules for (one) times. The pTREs are not enumerated according to length as it is in trivial enumeration, but in the number of recursive steps from the start $\phi = \heartsuit$. Once reaching a k-length closed pTRE, we check whether it is a $p\text{TRE}_+$.

During the recursive generation, if the current pTRE is an edge pTRE (see the definition in Section VI-A), instead of utilizing Rule $\heartsuit \rightarrow \langle\underline{\sigma}\rangle_{\mathbb{I}}$ directly, we replace the only placeholder with a special formula $\langle\Sigma^*\rangle_{[0,\infty)}$, and thus the resulting closed pTRE can be viewed as an over-approximation of all possible children of the current edge pTRE. This adapts the idea in [14] to our TRE settings. If this over-approximation fails to accepted all positive examples, we can conclude that all children of the edge pTRE are impossible to be $p\text{TRE}_+$, and therefore we prune the edge pTRE directly. The whole process is presented in Algorithm 3. We have the following results.

Theorem 2. A $p\text{TRE}_+$ generated after k steps is of length k. All k-length $p\text{TRE}_+$ are generated at the k-th step in the recursive generation.

Example 4. Consider the set of positive examples $\{\omega_1 = (a, 1.2)(a, 2.2)$, $\omega_2 = (b, 1.2)(a, 2.6)\}$. The edge pTRE $\langle\underline{a}_{\mathbb{I}} \cdot \heartsuit\rangle_{\mathbb{I}}$ can be pruned before substituting the last placeholder, since $(\underline{a}_{\mathbb{I}} \cdot \langle\Sigma^*\rangle_{[0,\infty)})_{\mathbb{I}}$ doesn't accept ω_2. $((\underline{a}_{\mathbb{I}} \vee \underline{b}_{\mathbb{I}})_{\mathbb{I}} \cdot \heartsuit)_{\mathbb{I}}$ can not be pruned using the above strategy, since $\langle\langle\underline{a}_{\mathbb{I}} \vee \underline{b}_{\mathbb{I}})_{\mathbb{I}} \cdot \langle\Sigma^*\rangle_{[0,\infty)}\rangle\rangle_{\mathbb{I}}$ accept both ω_1 and ω_2.

Containment Pruning. The recursive-style enumeration is more friendly to pruning. However, it brings a disadvantage of repeatedly generating pTREs. For instance, $((\underline{a}_{\mathbb{I}} \cdot \heartsuit)_{\mathbb{I}} \cdot (\heartsuit \cdot \underline{a}_{\mathbb{I}})_{\mathbb{I}})_{\mathbb{I}}$ can be generated directly from $((\underline{a}_{\mathbb{I}} \cdot \heartsuit)_{\mathbb{I}} \cdot (\heartsuit \cdot \heartsuit)_{\mathbb{I}})_{\mathbb{I}}$ and $((\heartsuit \cdot \heartsuit)_{\mathbb{I}} \cdot (\heartsuit \cdot \underline{a}_{\mathbb{I}})_{\mathbb{I}})_{\mathbb{I}}$. We introduce a strategy, named containment pruning, of which the insight is to remember the pruned pTREs by the edge pruning and to prevent generating their children from other pTREs. The pruned pTREs are stored in a set S_{doomed}. When a new pTRE is generated, it will be checked whether it is contained by a pTRE in S_{doomed}. The containment can be decided utilizing the methods proposed in [45], [46], [47]. If so, the new pTRE is pruned from the recursive generation and added to S_{doomed}. The whole process is presented in Algorithm 4.

Remark 1. The containment problem is of PSPACE-complete [45]. Hence, it is an expensive strategy. When and how frequently it should be utilized still needs to be explored.

C. Encoding and solving time restrictions

Suppose that we have a $p\text{TRE}_+$ ϕ that accepts all possible examples. In this section, we present an SMT-based method to filter all negative examples by synthesizing a set of instances of the parametric intervals in ϕ. For a negative example $\omega_- \in \Omega_-$, there are two situations w.r.t ϕ as follows. If $\omega_- \notin [\![\phi]\!]$, we do nothing. If $\omega_- \in [\![\phi]\!]$, it means that the untimed sequence $\mu(\omega_-)$ matches the corresponding RE $\mu(\phi)$, and we need to find a set of instances so that the resulting TRE φ can reject it. We also need to keep φ accepting all positive examples with these instances. If we can not find such a satisfiable TRE φ, we drop this ϕ and try the next candidate $p\text{TRE}_+$.

Let us start with an illustrative example for a negative example $\omega_- \in \Omega_-$ such that $\omega_- \in [\![\phi]\!]$.

Example 5. Consider a negative example $\omega_- = (a, 1.5)(b, 2)(b, 3)$ and a $p\text{TRE}_+$ $\phi = \langle\langle\underline{a}\rangle_{\mathbb{I}_4} \vee \langle\langle\underline{a}\rangle_{\mathbb{I}_7} \cdot \langle\underline{b}\rangle_{\mathbb{I}_8}\rangle_{\mathbb{I}_5}\rangle_{\mathbb{I}_2} \cdot \langle\langle\underline{b}\rangle^*_{\mathbb{I}_6}\rangle_{\mathbb{I}_3}\rangle_{\mathbb{I}_1}$, we have the untimed version $\mu(\omega_-) = abb$ and $\mu(\phi) = (a \vee ab) \cdot b^*$. It is obvious that $\mu(\omega_-)$ matches $\mu(\phi)$ with two parsing solutions. One is that ω is composed with the symbol 'a' at left of $\vee$, then 'b^*' repeats for 2 times. The other is 'ab' followed by one iteration of 'b^*'. Therefore, if we need the resulting TRE φ to reject ω_-, the instances of parametric intervals should reject both parsing solutions by filtering the timing information in ω_-.

Algorithm 4: Recursive Enumeration with Edge Pruning and Containment Pruning

input : required length k; alphabet Σ
output: The set of all k-length $p\text{TRE}_+$ S_+.

$S_0 \leftarrow \heartsuit$; $step \leftarrow 0$;
while $step < k$ **do**
  for $\phi \in S_0$ **do**
    if ϕ *is not closed* **then**
      S_1.add($\phi's$ direct children);
  $step \leftarrow step + 1$;
  for $\phi \in S_1$ **do**
    if ϕ *is an edge pTRE* **then**
      do edge pruning on ϕ;
      if ϕ *is pruned* **then**
        S_{doomed}.add(ϕ);
    do RE containment pruning on p;
    if ϕ *is pruned* **then**
      S_{doomed}.add(ϕ);
  $S_0 \leftarrow S_1$;
  $S_1 \leftarrow \{\}$;
foreach $\phi \in S_0$ **do**
  if ϕ *is closed* **then**
    flag $\leftarrow$ check_acceptable(ϕ,Ω_+);
    if *flag* $= \top$ **then**
      S_+.add(ϕ);
return S

Hence, we need to find all parsing solutions of $\mu(\omega_-)$ w.r.t the untimed RE $\mu(\phi)$. For our particular encoding problem, the basic idea is as follows.

1) Label each symbol in $\mu(\phi)$ with a subscript which is the position of the corresponding node on the syntax tree, thus obtaining a linear expression ψ.
2) Transform ψ to an nondeterministic finite automata (NFA) utilizing Glushkov's construction [48], which is to construct an NFA from a RE. In a Glushkov's NFA, each transition action is labeled by a subscripted symbol of a leaf node of the syntax tree. Each *accepting path* (defined below) of the NFA, by taking the labels of all transitions of an into a sequence, represents how the symbols of an untimed word are matched to the leaf nodes of the tree.
3) Find all parsing solutions w.r.t $\mu(\omega_-)$ and $\mu(\phi)$, each is represented as an accepting path.
4) For each path p, we encode the requirement that the time information in ω_- does not satisfy the time restrictions along the path into an SMT formula γ.

An accepting path p of the Glushkov's NFA for ϕ is a finite sequence of pairs (x, d), where x is the subscript of a symbol in ϕ and d is the depth of the corresponding node on the syntax tree. We write $p[i]$ for the element at the i-th index of p.

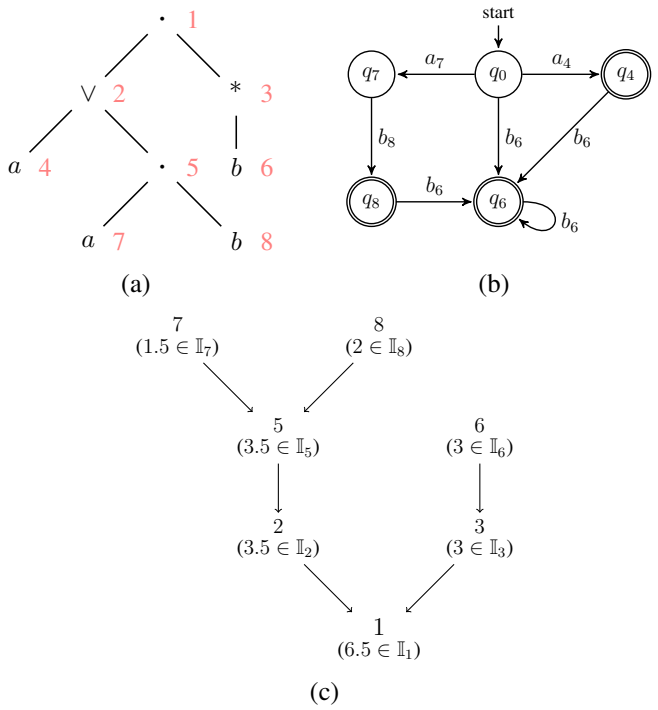

Fig. 3. (a): the syntax tree of $\mu(\phi) = (a \vee ab) \cdot b^*$; (b): the corresponding Glushkov's NFA A of $\psi = (a_4 \vee a_7 b_8) \cdot b_6^*$; (c): the encoding process of path $7, 8, 6$.

Example 6. Continue Example 5. Given $\mu(\phi) = (a \vee ab) \cdot b^*$, we have the corresponding syntax tree in Fig. 3. By renaming the symbols, we get a linear expression $\psi = (a_4 \vee a_7 b_8) \cdot b_6^*$. The subscript is the position of the corresponding symbols on the syntax tree. Then we build a Glushkov's NFA A w.r.t ψ as shown in Fig. 3. For the string $\mu(\omega_-) = abb$, two accepting paths $(4, 3), (6, 3), (6, 3)$ and $(7, 4), (8, 4), (6, 3)$ are found. For each path, based on the semantics of TRE, we can encode the requirement that the corresponding parametric intervals in ϕ filter the path.

Algorithm 5 depicts the process to encode the requirement that a $p\text{TRE}_+$ ϕ rejects an accepting path p w.r.t a negative example ω_- into an SMT formula γ. Note that to encode the requirement that a positive example ω_+ is accepted by ϕ is similar, except that at the end of the algorithm it is the disjunction of all β instead of the conjunction of $\neg\beta$ that is constructed. Function parent(ϕ, p, i) returns a pair (x', d'), where x' is the subscript of $\phi[p[i].x].parent$ and $d' = p[i].d - 1$. And function encode_single_node($ctimes$, p, i) is to generate constraint $ctimes[i] \in \mathbb{I}_{p[i].x}$. At first each symbol of ω_- is matched to a leaf node on the syntax tree of ϕ and the time constraints of these leaf nodes are encoded. Then after each loop the nodes with the max depth in the array $cnode$ are replaced by their parent nodes on the tree, and the time constraints of these parent nodes are encoded.

Example 7. Continue to consider $\omega_- = (a, 1.5)(b, 2)(b, 3)$, $\phi = \langle\langle\langle\underline{a}\rangle_{\mathbb{I}_4} \vee \langle\langle\underline{a}\rangle_{\mathbb{I}_7} \cdot \langle\underline{b}\rangle_{\mathbb{I}_8}\rangle_{\mathbb{I}_5}\rangle_{\mathbb{I}_2} \cdot \langle\langle\underline{b}\rangle^*_{\mathbb{I}_6}\rangle_{\mathbb{I}_3}\rangle_{\mathbb{I}_1}$, and one accepting path $p = (7, 4), (8, 4), (6, 3)$. We have $ctimes = 1.5, 2, 3$ at

Algorithm 5: Encoding path formula

input : a negative example ω_-; a pTRE$_+$ ϕ; an accepting path p.
output: a path formula γ.

1 $ctimes \leftarrow \lambda(\omega)$; $depth \leftarrow max\_d(p)$, $C \leftarrow \emptyset$;
2 **for** $i \in [0, |p|-1]$ **do**
3 $\beta \leftarrow$ encode_single_node($ctimes$, p, i); C.add(β);
4 **while** $depth \neq 1$ **do**
5 $i \leftarrow 0$; $next\_p \leftarrow []$; $next\_time \leftarrow []$;
6 **while** $i < |p|$ **do**
7 $cpos \leftarrow p[i].x$;
8 $cnode \leftarrow \phi[cpos]$;
9 $next\_pos \leftarrow p[i+1].x$;
10 $next\_node \leftarrow \phi[next\_pos]$;
11 **if** $cnode.depth = depth$ **then**
12 **if** *cnode.parent is a disjunction node* **then**
13 $next\_p$.add($cnode.parent$);
14 $next\_time$.add($ctimes[i]$);
15 $i \leftarrow i+1$;
16 **if** *cnode.parent is a concatenation node* ***and*** $next\_node.parent = cnode.parent$ **then**
17 $next\_p$.add($cnode.parent$);
18 $next\_time$.add($ctimes[i] + ctimes[i+1]$);
19 $i \leftarrow i+1$;
20 **if** *cnode.parent is a star node* **then**
21 **find** the first $j > i$ s.t. $\phi[p[j].x]$.parent$\neq cnode$.parent **or** $j = |p|$;
22 $next\_p$.add($cnode.parent$);
23 $next\_time$.add($\sum_{i \le k < j} ctime[k]$);
24 $i \leftarrow j$;
25 $\beta \leftarrow$encode_single_node($ctimes$, p, i);
26 C.add(β);
27 **else**
28 $next\_p.add(p[i])$;
29 $next\_time.add(ctime[i])$;
30 $i \leftarrow i+1$;
31 $p \leftarrow next\_p$;
32 $ctimes \leftarrow next\_time$;
33 $depth \leftarrow max\_d(p)$;
34 $\gamma \leftarrow \bigwedge_{\beta \in C} \neg\beta$;
35 **return** γ ;

the beginning. After encoding the timing constraints for leaf nodes at the positions $7, 8, 6$ (Line 3), we obtain the formulas $1.5 \in \mathbb{I}_7$, $2 \in \mathbb{I}_8$, and $3 \in \mathbb{I}_6$. Since the father node of the two nodes at position 7 and 8 is the concatenation node at position 5, we have $next\_p = (5,3),(6,3)$ and $next\_ctimes = 3.5, 3$, and thereby obtain a new formula $3.5 \in \mathbb{I}_5$ (Line 17). Since the father node of the node at position 5 is the disjunction node at position 2, we have $next\_p = (2,2),(6,3)$ and $next\_ctimes = [3.5, 3]$, and obtain a new formula $3.5 \in \mathbb{I}_2$ (Line 13). With the same process, we obtain the formulas $3 \in \mathbb{I}_3$ and $6.5 \in \mathbb{I}_1$ for the time restrictions of the nodes at positions 3 and 1. Finally, we get a path formula $\gamma = \neg(1.5 \in \mathbb{I}_7 \wedge 2 \in \mathbb{I}_8 \wedge 3 \in \mathbb{I}_6 \wedge 3.5 \in \mathbb{I}_5 \wedge 3.5 \in \mathbb{I}_2 \wedge 3 \in \mathbb{I}_3 \wedge 6.5 \in \mathbb{I}_1)$.

In general, the basic encoding process of the SMT problem Φ_Ω with pTRE$_+$ ϕ and examples Ω is as follows:

- Positive examples encoding: For each positive example ω_+, we first solve the parsing problem whether $\omega_+ \in [\![\phi]\!]$, and find all accepting paths. Once the paths are identified, encode the time constraints associated with each path into formulas. Subsequently, construct the disjunction of path formulas to form the constraints for a single positive example.
- Negative examples encoding: For each negative example ω_-, we first check whether it is consistent with pTRE$_+$ ϕ. If not, then we do not need to parse it. If it is consistent, proceed to solve the parsing problem $\omega_- \in [\![\phi]\!]$? and find all accepting paths. Once the paths are identified, encode the time constraints associated with each path into formulas. Subsequently, construct the conjunction of the negations of all path formulas to form the constraints for a single negative example.
- SMT problem construction: Φ_Ω is the conjunction of all constraints encoded from all positive and negative examples.

Theorem 3. Given a pTRE$_+$ ϕ, there exists a TRE $\varphi \in [\phi]$ consistent with Ω if and only if Φ_Ω is satisfiable.

VII. Experiments and Implementation.

We have implemented our synthesis methods proposed in Section VI, including the strategies for enumerating and pruning pTRE as proposed in Section VI-B along with the timed constraints encoding method detailed in Section VI-C. The prototype is developed in Python 3.8. We use Z3 [49] as our SMT solver. All experiments were conducted on a laptop:

- Processor: 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz, with 4 cores and 8 logical processors.
- RAM: 16GB installed

Note that the correctness of our synthesis method is guaranteed by the proposed theorems. Therefore, we conducted three kinds of experiments to demonstrate its effectiveness and efficiency. First, we compared the efficiency of the three synthesis strategies. Second, we demonstrated the scalability of our synthesis process on randomly generated timed words. At last, we evaluated our synthesis method using a case study involving a train gate system [50] modeled by a timed automaton. We utilized the timed automata uniform sampler **WORDGEN** [51] to generate timed words.

A. The Comparison of Strategies

In this subsection, we evaluate and compare the different strategies: recursive with edge pruning and recursive with containment pruning. The experiments were conducted using a setting of alphabet $\Sigma = \{a, b\}$ and time delay $t \in [0, 4)$. Timed words of selected lengths are randomly generated and fed into a one-clock and full-reset timed automaton (as depicted in Fig. 4). Accepted timed words are marked as positive examples, while rejected ones are marked as negative. This process is repeated until we get the desired *number of positive examples (nop)*. In the experiment, around 40% of the randomly generated timed words are marked positive.

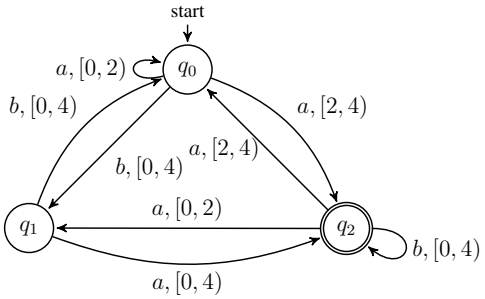

Fig. 4. The timed automaton used to mark positive and negative examples in the strategies comparison.

TABLE I
EXPERIMENTAL RESULTS FOR DIFFERENT STRATEGIES. **ABOVE**: TRIVIAL. **MIDDLE**: RECURSIVE & EDGE PRUNING. **BELOW**: RECURSIVE & CONTAINMENT PRUNING.

time(s) nop / length	5	7	9	11	13	15
6	0.61	146.3	0.38	295.6	173.5	167.8
7	47.3	169.6	169.4	161.4	170.0	172.2
8	1.75	48.9	194.1	165.9	173.6	171.3
9	160.5	1.76	172.1	169.6	170.1	167.5
10	156.1	169.7	175.8	172.4	160.3	176.7

time(s) nop / length	5	7	9	11	13	15
6	0.46	114.7	0.35	236.5	130.7	128.4
7	36.2	130.4	130.8	122.9	123.9	129.2
8	1.55	35.4	153.2	125.3	127.6	127.8
9	126.8	1.50	129.0	127.6	124.1	119.5
10	117.2	132.3	133.1	132.3	119.4	124.0

time(s) nop / length	5	7	9	11	13	15
6	0.41	107.8	0.33	245.9	125.8	122.5
7	32.6	128.5	125.7	120.1	115.0	120.6
8	1.52	32.1	154.7	119.2	124.9	118.8
9	122.5	1.48	124.3	122.4	117.5	120.1
10	112.7	127.0	127.2	124.8	116.4	123.2

As the randomly generated timed words are not enough to characterize the timed automaton, we don't require that the synthesized TRE exactly matches the corresponding TRE of the automaton.

Strategy setting:

- Edge pruning strategy: This strategy involves pruning only on edge pTREs.
- Containment Pruning Strategy: In this strategy, we perform a partial containment check. Specifically, we verify if the syntax tree of a longer pTRE contains that of a shorter one, excluding placeholders $\heartsuit$.

Table I presents the experimental results by using different strategies. As the length of the examples increases, longer examples tend to produce more complex TRE with minimal length. Consequently, the number of pTREs requiring enumeration and verification grows exponentially, leading to a significant increase in computational time. Additionally, the results indicate that pruning strategies generally demonstrate superior performance compared to the trivial strategy. Nonetheless, the implementation of more sophisticated pruning techniques does not necessarily result in enhanced search efficiency. This may be attributed to the increased time required for containment checks as the set of identified pTREs expands.

TABLE II
TREs USED IN SCALABILITY EXPERIMENT

Length of TRE	**Size of Alphabet** $\lvert\Sigma\rvert$	
	2	3
6	$\langle \underline{a}_{\mathbb{I}} \cdot \langle \underline{b}_{\mathbb{I}} \rangle^* \rangle_{\mathbb{I}} \cdot \underline{a}_{\mathbb{I}}$	$\langle \underline{a} \vee \underline{b} \rangle_{\mathbb{I}} \cdot \langle \underline{c}^* \rangle_{\mathbb{I}}$
7	$\langle \underline{a}_{\mathbb{I}} \cdot \underline{b}_{\mathbb{I}} \rangle \vee \langle \underline{b}_{\mathbb{I}} \cdot \underline{a}_{\mathbb{I}} \rangle$	$\langle \underline{a} \vee \underline{b} \rangle^*_{\mathbb{I}} \cdot \underline{c}^*_{\mathbb{I}}$
8	$\langle \underline{a}_{\mathbb{I}} \cdot \underline{b}_{\mathbb{I}} \rangle^* \cdot \langle \underline{a} \vee \underline{b} \rangle_{\mathbb{I}}$	$\langle \underline{a}_{\mathbb{I}} \cdot \underline{b}_{\mathbb{I}} \cdot \underline{c}_{\mathbb{I}} \rangle^* \cdot \underline{a}_{\mathbb{I}}$
9	$\langle \underline{a}^* \rangle_{\mathbb{I}} \cdot \langle \underline{b}^* \rangle_{\mathbb{I}} \cdot \langle \underline{a} \cdot \underline{b} \rangle_{\mathbb{I}}$	$\langle \underline{a} \cdot \underline{b} \rangle_{\mathbb{I}} \cdot \langle \langle \underline{c} \cdot \underline{b} \rangle_{\mathbb{I}} \vee \underline{b}_{\mathbb{I}} \rangle$

Length of TRE	**Size of Alphabet** $\lvert\Sigma\rvert$	
	4	5
6	$\langle \langle \underline{a}^* \vee \underline{b} \rangle_{\mathbb{I}} \cdot \underline{c} \rangle_{\mathbb{I}}$	$\langle \underline{a}_{\mathbb{I}} \rangle^*_{\mathbb{I}} \cdot \underline{b}_{\mathbb{I}} \vee \underline{c}_{\mathbb{I}_{\mathbb{I}}}$
7	$\langle \underline{a}_{\mathbb{I}} \cdot \underline{b}_{\mathbb{I}} \rangle_{\mathbb{I}} \vee \langle \underline{c}_{\mathbb{I}} \cdot \underline{a}_{\mathbb{I}} \rangle$	$\langle \underline{a}_{\mathbb{I}} \vee \underline{b}_{\mathbb{I}} \cdot \underline{c} \rangle_{\mathbb{I}} \cdot \underline{a}_{\mathbb{I}}$
8	$\underline{a}_{\mathbb{I}} \cdot \underline{b}_{\mathbb{I}} \cdot \langle \underline{c} \vee \underline{d} \rangle^*_{\mathbb{I}}$	$\langle \langle \underline{a} \vee \underline{b} \rangle^*_{\mathbb{I}} \rangle_{\mathbb{I}} \cdot \langle \underline{c} \vee \underline{d} \rangle_{\mathbb{I}}$
9	$\langle \underline{a} \vee \underline{b} \rangle^*_{\mathbb{I}} \cdot \langle c \vee \underline{d} \rangle^*_{\mathbb{I}}$	$\langle \underline{a}_{\mathbb{I}} \cdot \underline{b} \cdot \underline{c} \rangle_{\mathbb{I}} \cdot \langle \underline{d}_{\mathbb{I}} \cdot \underline{e} \rangle_{\mathbb{I}}$

B. Scalability

In this subsection, we evaluate the scalability of our method. We prepare 16 TREs of which the length L is scaled from 6 to 9 and the size of alphabet $|\Sigma|$ is scaled from 2 to 5, as listed in Table II. For conciseness and readability, the time intervals in TREs are abbreviated as "$\mathbb{I}$", and brackets "$\langle\rangle$" containing a single letter are omitted in the table. In experiment, each "$\mathbb{I}$" is randomly assigned with $(a, b]$, where $0 \leq a < b \leq 10, a, b \in \mathbb{N}$, to form valid TREs. Then we transform these TREs into corresponding TAs. Positive examples are sampled directly from these TAs, and negative ones are sampled from complemented TAs. The maximum of the upper bounds of guards in complemented TAs is 15. We use **WORDGEN** to generate examples. The range of example lengths is $l \leq 8, 11, 14$ (some of the TREs accept only timed words shorter than 8, so we only test them in group $l \leq 8$). For each TRE with a different length of examples, we conducted 3 trials. And for each trial, 300 positive and 300 negative examples were sampled. The time limit of each trial is 1800 seconds.

As shown in Table III, our method is capable of synthesizing a TRE in a reasonable time. Along with a larger alphabet, a more complex target TRE, and longer examples, the more time it costs. The size of the alphabet and the length of the target TRE influence the running time hugely since they increase the number of pTREs to be enumerated. Length of examples exerts a slight influence on the duration of the synthesis process.

Note that the synthesis results may not be the same as the target TREs. Sometimes only the time intervals are different, but it is more often that a synthesized TRE is shorter than the corresponding target TRE. This phenomenon can be attributed to the synthesis method always synthesizing the minimal consistent TRE φ. If there exists a TRE φ containing the language of the target TRE ϕ, i.e., $[\![\varphi]\!] \supset [\![\phi]\!]$, our method

TABLE III
AVERAGE TIME EXPANSE (SECONDS) IN SCALABILITY EXPERIMENT.
ABOVE: LENGTH OF EXAMPLES $l \leq 8$. **MIDDLE**:$l \leq 11$. **BELOW**: $l \leq 14$.

Length of TRE	Size of Alphabet $\|\Sigma\|$			
	2	3	4	5
6	3.8	6.2	17.3	34.1
7	26.6	37.4	60.1	95.2
8	173.0	316.5	502.7	772.8
9	302.5	513.6	871.8	1365

Length of TRE	Size of Alphabet $\|\Sigma\|$			
	2	3	4	5
6	4.2	7.7	22.4	46.7
7	-	46.2	-	-
8	198.6	342.7	374.7[a]	406.9[a]
9	363.8	-	591.3[a]	-

Length of TRE	Size of Alphabet $\|\Sigma\|$			
	2	3	4	5
6	4.8	9.7	27.2	63.0
7	-	58.5	-	-
8	236.5	170.6[a]	419.3[a]	157.5[a]
9	392.5	-	662.8[a]	-

[a] Not all 3 trials successfully generate the exact target TRE.

TABLE IV
AVERAGE TIME EXPANSE (SECONDS) AND CORRECT TIMES IN INPUT EXPERIMENT.

Scale $(L \times \|\Sigma\|)$	Number of positive examples nop			
	300	450	600	750
$(7,3)$	48.4/3	67.5/3	88.3/3	113.6/3
$(7,4)$	85.3/3	124.7/3	152.1/3	214.9/3
$(8,3)$	334.5/3	452.2/3	581.0/3	794.7/3
$(8,4)$	370.4/2	489.6/2	834.5/3	1041/3
$(9,3)$	465.2/2	681/2	852.8/3	1093/3
$(9,4)$	603.7/1	824.6/1	1317/2	1560/2

will find such φ. The shorter TRE φ is discarded if and only if the set of negative examples contains a member that is recognized by the shorter TRE but not by the target TRE, i.e. $\exists \omega_- \in \Omega_-, \omega_- \in [\![\varphi]\!] \setminus [\![\phi]\!]$. The likelihood of encountering a longer negative example or a negative example sampled with larger alphabet such that $\omega_- \in [\![\varphi]\!] \setminus [\![\phi]\!]$ is smaller. Consequently, when the number of examples is fixed, it is less probable that the target TRE ϕ will be synthesized as the length of examples and size of alphabet increase.

We also conducted experiments to evaluate the number of examples needed to synthesize a TRE exactly matching the target one. We selected the TREs in table II with $L \in [7,9]$, alphabet size $|\Sigma| \in [3,4]$ and chose range of example length $l \leq 11$. As shown in Table III, 300 examples are not enough to synthesize a TRE exactly matching the target TRE. We extended the number of the positive examples $nop = 300, 450, 600, 750$, and did the same for the negative example set. For each group, we still conducted 3 trials. Table IV presents the average running time and the number of trials generating matching TRE. For example, "48.3/3" means that the average running time is 48.3 seconds and all 3 trials are successful. The results show that as the length of TRE and the size of the alphabet increase, the number of examples required for a correct synthesis result increases significantly.

TABLE V
THE TEMPLATE pTREs IN SCHEDULING CASE STUDY.

No.	Template pTRE	States	Parameters
1	$\langle \underline{a_2}_{\mathbb{I}_1} \cdot \underline{a_1}_{\mathbb{I}_2} \rangle_{\mathbb{I}_3} \cdot \underline{b_1}$ $\cdot \langle \langle \underline{a_2} \cdot \underline{b_2} \rangle_{\mathbb{I}_4} \rangle^* \cdot \underline{b_2}$	$q_0, q_2,$ q_3, q_5	$p_1, p_2,$ $D_2 - D_1$
2	$\langle \underline{a_1}_{\mathbb{I}_1} \cdot \underline{b_1} \rangle \vee \langle \underline{a_2}_{\mathbb{I}_2} \cdot \underline{b_2} \rangle$ $\vee \langle \langle \underline{a_2}_{\mathbb{I}_3} \cdot \underline{a_1} \rangle_{\mathbb{I}_4} \cdot \underline{b_2} \cdot \underline{b_1} \rangle$	$q_0, q_1,$ q_2, q_4	$p_1, p_2,$ $D_2 - D_1$
3	$\langle \underline{a_1}_{\mathbb{I}_1} \cdot \underline{b_1} \rangle \vee \langle \underline{a_2}_{\mathbb{I}_2} \cdot \underline{b_2} \rangle \vee \langle \underline{a_1}_{\mathbb{I}_3} \rangle$ $\cdot \underline{a_2}_{\mathbb{I}_4} \cdot \langle \langle \underline{b_1} \cdot \underline{a_1} \rangle^*_{\mathbb{I}_5} \rangle_{\mathbb{I}_6} \cdot \underline{b_1} \cdot \underline{b_2}$	$q_0, q_1,$ q_2, q_3	$p_1, p_2,$ $D_2 - D_1$

TABLE VI
THE RUNNING TIME (SECONDS) FOR TEMPLATE pTREs.
ABOVE: TEMPLATE 1. **MIDDLE**: TEMPLATE 2. **BELOW**: TEMPLATE 3.

Length of Examples l	Number of positive examples nop			
	300	450	600	750
≤ 11	42.4	56.3	73.8	95.5
≤ 15	52.3	75.2	98.5	130.6
≤ 19	59.6	84.7	113.1	152.8

Length of Examples l	Number of positive examples nop			
	300	450	600	750
≤ 11	58.7	75.5	94.6	116.3
≤ 15	67.9	92.2	114.5	138.3
≤ 19	86.0	118.7	143.6	177.1

Length of Examples l	Number of positive examples nop			
	300	450	600	750
≤ 11	86.8[a]	122.5	157.4	183.6
≤ 15	117.3	146.2	174.7	203.2
≤ 19	132.8	169.0	204.3	241.5

[a] Not successfully generate all the target intervals.

C. Case study

The first case study is to continue the illustrative example given in Section II. For the gray-box setting, we can build the model with parameters directly as we did in Section II. Therefore, we aim to see if our synthesis approach helps to extract the parameters from sampled traces. For sampling traces, the parameters of the case are set as: minimum interarrival times $p_1 = 8$, $p_2 = 14$, execution times $C_1 = 4$, $C_2 = 6$, and deadlines $D_1 = 8$, $D_2 = 20$.

We predefine three templates pTREs as listed in Table V, each capturing a part of the automaton. We sample these templates pTREs with all time intervals to be $(0, 30]$. For instance, the 4 target intervals in Template 1 are $\mathbb{I}_1 = [p_2, 30]$, $\mathbb{I}_2 = (0, D_2 - D_1]$, $\mathbb{I}_3 = [p_1, 30]$, $\mathbb{I}_4 = [p_2, 30]$. After sampling on the templates, we select the traces that can be accepted by the TA as positive examples, and the remaining as negative examples. Then we try to synthesize time intervals directly from these templates, skipping the enumeration of $p\text{TRE}_+$. We range the length of examples to $l \leq 11, 15, 19$, the number of examples to $nop = 300, 450, 600, 750$. We

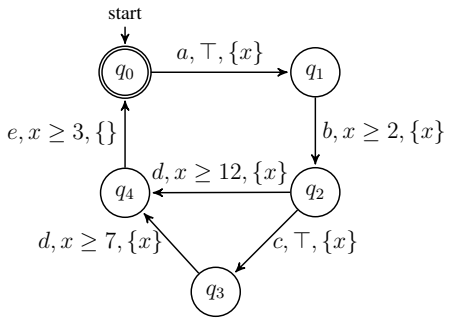

Fig. 5. Train TA

TABLE VII
THE RUNNING TIME (SECONDS) OF SYNTHESIZING TRE IN TRAIN TA EXPERIMENT.

Length of Examples l	**Number of positive examples** nop			
	300	450	600	750
≤ 11	3347	3756	6882/★	TO
≤ 15	1825	3912	4458	TO
≤ 19	1772	1946	4751	5239

★ indicates the resulted TRE exactly matches the target TRE φ_{Train}; TO represents a timeout.

conducted one trial for each template with each setting of (l, nop). Table VI presents the synthesis for each trial. In most cases (except for the case $l \leq 11$ and $nop = 300$), our method successfully learned the parameters involved. For instance, it takes 42.4 seconds to get the time intervals $\mathbb{I}_1 = [14, 30]$, $\mathbb{I}_2 = (0, 12]$, $\mathbb{I}_3 = [8, 30]$, and $\mathbb{I}_4 = [14, 30]$ for Template 1 with $l \leq 11, nop = 300$. So we can conclude from the synthesis results that $p_1 = D_1 = 8$, $p_2 = 14$, $D_2 - D_1 = 12$, and $D_2 = 20$. Then, according to EDF, we know that if we can guarantee the execution times C_1, C_2 satisfy $\frac{C_1}{8} + \frac{C_2}{14} \leq 1$, then the system is schedulable.

The second case study is about a train gate simplified from [50], which is modeled by the TA as depicted in Fig. 5, where a, b, c, d, e stand for "starts", "train approaches", "train halts", "train enters the gate", and "train leaves", respectively. The corresponding TRE of this TA is $\varphi_{Train} = \langle \underline{a} \cdot \langle \underline{b} \rangle_{[2,\infty)} \cdot \langle \langle \underline{d} \rangle_{[12,\infty)} \vee \langle \underline{c} \cdot \langle \underline{d} \rangle_{[7,\infty)} \rangle \rangle \cdot \langle \underline{e} \rangle_{[3,\infty)} \rangle^*$. It is more complex than the ones in previous subsections with length $L = 12$ and alphabet size $|\Sigma| = 5$. We set the range of examples length to $l \leq 11, 15, 19$, the number of examples to $nop = 300, 450, 600, 750$, and the maximum delay time to 20 (substituting for ∞ as time upper bounds). We conducted one trial for each setting with a time limit of 7200 seconds. Table VII presents the running time for each trial. In most cases, our method terminates and returns the minimal TRE consistent with the randomly generated examples. What's more, a trial even returns a TRE exactly matching the target TRE. It shows the effectiveness and efficiency of our method in practice.

VIII. CONCLUDING REMARKS

We addressed the TRE synthesis problem by theoretically establishing its decidability and proposing a practical framework aimed at finding a minimal TRE recognizing the given set of real-time behaviors. The framework is based on the insight of first finding a pTRE that accepts all positive examples and then appropriately deciding its time intervals. We have implemented the synthesis method, along with three strategies for enumerating and pruning pTREs. The three strategies were compared experimentally and the framework's scalability was demonstrated through tests on uniformly sampled real-time behaviors of TAs, which are from TREs of various complexities.

Note that, as we mentioned in Section III, although (generalized extended) timed regular expressions have the same expressive power as timed automata [1], we only consider (extended) timed regular expressions in this work, excluding the renaming function (θ) and the conjunction ($\wedge$). Therefore, our method may not apply to all real-time scheduling models, which are beyond extended timed regular expressions. Which kinds of timed automata have the same expressive power as (extended) timed regular expressions remains unclear.

In future work, we plan to explore other enumeration and pruning strategies for our framework. The challenge lies in striking a balance between reducing the effort required to check pTREs and expanding the scope of pruning effectively. Additionally, we intend to identify the sufficient conditions for positive (and negative) examples required by our synthesis framework. Furthermore, we are to provide a PAC bound regarding the number of samples needed to achieve sufficient conditions of examples incorporating the uniform sampling technique. Which kinds of scheduling models can be expressed by (extended) timed regular expressions is also an interesting research topic.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments, which significantly improved the quality of this paper. Ziran Wang, Jie An, and Naijun Zhan are supported by the National Key R&D Program of China under grant No. 2022YFA1005101, the Natural Science Foundation of China (NSFC) under grants No. W2511064, No. 62192732 and No. 62032024, and the ISCAS Basic Research Grant No. ISCAS-JCZD202406. Miaomiao Zhang is supported by the NSFC Grant No. 62472316. Zhenya Zhang is supported by the JSPS Grant No. JP25K21179.

REFERENCES

[1] E. Asarin, P. Caspi, and O. Maler, "Timed regular expressions," *J. ACM*, vol. 49, no. 2, pp. 172–206, 2002.
[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.
[3] Y. Tang, N. Guan, and W. Yi, "Real-time task models," in *Handbook of Real-Time Computing*, Y. Tian and D. C. Levy, Eds. Springer, 2022, pp. 469–487.
[4] P. Ekberg and W. Yi, "Complexity of uniprocessor scheduling analysis," in *Handbook of Real-Time Computing*, Y. Tian and D. C. Levy, Eds. Springer, 2022, pp. 489–506.

[5] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*. IEEE Computer Society, 2011, pp. 71–80.

[6] D. Ulus, T. Ferrère, E. Asarin, and O. Maler, "Timed pattern matching," in *FORMATS 2014*, ser. LNCS, vol. 8711. Springer, 2014, pp. 222–236.

[7] M. Waga, T. Akazaki, and I. Hasuo, "A boyer-moore type algorithm for timed pattern matching," in *FORMATS 2016*, ser. LNCS, vol. 9884. Springer, 2016, pp. 121–139.

[8] M. Waga and I. Hasuo, "Moore-machine filtering for timed and untimed pattern matching," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2649–2660, 2018.

[9] M. Waga, É. André, and I. Hasuo, "Parametric timed pattern matching," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 10:1–10:35, 2023.

[10] L. Schmidt, A. Narayan, and S. Fischmeister, "TREM: a tool for mining timed regular specifications from system traces," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 901–906.

[11] A. Narayan, G. Cutulenco, Y. Joshi, and S. Fischmeister, "Mining timed regular specifications from system traces," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, pp. 46:1–46:21, 2018.

[12] D. Angluin, "On the complexity of minimum inference of regular sets," *Inf. Control.*, vol. 39, no. 3, pp. 337–350, 1978.

[13] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *POPL 2011*. ACM, 2011, pp. 317–330.

[14] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," in *GPCE 2016*. ACM, 2016, pp. 70–80.

[15] R. Pan, Q. Hu, G. Xu, and L. D'Antoni, "Automatic repair of regular expressions," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 139:1–139:29, 2019.

[16] N. Kushman and R. Barzilay, "Using semantic unification to generate regular expressions from natural language," in *HLT-NAACL 2013*. The Association for Computational Linguistics, 2013, pp. 826–836.

[17] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay, "Neural generation of regular expressions from natural language with minimal domain knowledge," in *EMNLP 2016*. The Association for Computational Linguistics, 2016, pp. 1918–1923.

[18] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J. Lou, T. Liu, and D. Zhang, "Semregex: A semantics-based approach for generating regular expressions from natural language specifications," in *EMNLP 2018*. Association for Computational Linguistics, 2018, pp. 1608–1618.

[19] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *PLDI 2020*, pp. 487–502.

[20] E. M. Gold, "Complexity of automaton identification from given data," *Inf. Control.*, vol. 37, no. 3, pp. 302–320, 1978.

[21] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," *Inf. Comput.*, vol. 103, no. 2, pp. 299–347, 1993.

[22] R. Parekh and V. G. Honavar, "Learning DFA from simple examples," *Mach. Learn.*, vol. 44, no. 1/2, pp. 9–35, 2001.

[23] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[24] O. Grinchtein, B. Jonsson, and M. Leucker, "Learning of event-recording automata," *Theoretical Computer Science*, vol. 411, no. 47, pp. 4029–4054, 2010.

[25] L. Henry, T. Jéron, and N. Markey, "Active learning of timed automata with unobservable resets," in *FORMATS 2020*, ser. LNCS, vol. 12288. Springer, 2020, pp. 144–160.

[26] S. Verwer, M. de Weerdt, and C. Witteveen, "Efficiently identifying deterministic real-time automata from labeled data," *Mach. Learn.*, vol. 86, no. 3, pp. 295–333, 2012.

[27] J. Schmidt, A. Ghorbani, A. Hapfelmeier, and S. Kramer, "Learning probabilistic real-time automata from multi-attribute event logs," *Intell. Data Anal.*, vol. 17, no. 1, pp. 93–123, 2013.

[28] L. Cornanguer, C. Largouët, L. Rozé, and A. Termier, "TAG: learning timed automata from logs," in *36th AAAI Conference on Artificial Intelligence, AAAI 2022*. AAAI Press, 2022, pp. 3949–3958.

[29] J. An, L. Wang, B. Zhan, N. Zhan, and M. Zhang, "Learning real-time automata," *Sci. China Inf. Sci.*, vol. 64, no. 9, 2021.

[30] J. An, B. Zhan, N. Zhan, and M. Zhang, "Learning nondeterministic real-time automata," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, pp. 99:1–99:26, 2021.

[31] S. Verwer, M. de Weerdt, and C. Witteveen, "The efficiency of identifying timed automata and the power of clocks," *Inf. Comput.*, vol. 209, no. 3, pp. 606–625, 2011.

[32] J. An, M. Chen, B. Zhan, N. Zhan, and M. Zhang, "Learning one-clock timed automata," in *TACAS 2020*, ser. Lecture Notes in Computer Science, vol. 12078. Springer, 2020, pp. 444–462.

[33] R. Xu, J. An, and B. Zhan, "Active learning of one-clock timed automata using constraint solving," in *the 20th International Symposium on Automated Technology for Verification and Analysis, ATVA 2022*, ser. LNCS, vol. 13505. Springer, 2022, pp. 249–265.

[34] S. Dierl, F. M. Howar, S. Kauffman, M. Kristjansen, K. G. Larsen, F. Lorber, and M. Mauritz, "Learning symbolic timed models from concrete timed data," in *NFM 2023*, ser. LNCS, vol. 13903. Springer, 2023, pp. 104–121.

[35] F. W. Vaandrager, R. Bloem, and M. Ebrahimi, "Learning mealy machines with one timer," in *LATA 2021*, ser. LNCS, vol. 12638. Springer, 2021, pp. 157–170.

[36] M. Tappler, B. K. Aichernig, K. G. Larsen, and F. Lorber, "Time to learn - learning timed automata from tests," in *FORMATS 2019*, ser. LNCS, vol. 11750. Springer, 2019, pp. 216–235.

[37] B. K. Aichernig, A. Pferscher, and M. Tappler, "From passive to active: Learning timed automata efficiently," in *NFM 2020*, ser. LNCS, vol. 12229. Springer, 2020, pp. 1–19.

[38] M. Tappler, B. K. Aichernig, and F. Lorber, "Timed automata learning via SMT solving," in *NFM 2022*, ser. LNCS, vol. 13260. Springer, 2022, pp. 489–507.

[39] M. Waga, "Active learning of deterministic timed automata with myhill-nerode style characterization," in *CAV 2023*, ser. LNCS, vol. 13964. Springer, 2023, pp. 3–26.

[40] Y. Teng, M. Zhang, and J. An, "Learning deterministic multi-clock timed automata," in *Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2024, Hong Kong SAR, China, May 14-16, 2024*. ACM, 2024, pp. 6:1–6:11.

[41] K. Watanabe, G. Fainekos, B. Hoxha, M. Lahijanian, D. V. Prokhorov, S. Sankaranarayanan, and T. Yamaguchi, "Timed partial order inference algorithm," in *ICAPS 2023*. AAAI Press, 2023, pp. 639–647.

[42] A. R. Meyer and L. J. Stockmeyer, "The equivalence problem for regular expressions with squaring requires exponential space," in *13th Annual Symposium on Switching and Automata Theory, Maryland, USA, October 25-27, 1972*. IEEE Computer Society, 1972, pp. 125–129.

[43] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, 2021, vol. 336, pp. 1267–1329.

[44] Z. Wang, J. An, N. Zhan, M. Zhang, and Z. Zhang, "On synthesis of timed regular expressions," 2025, arXiv:2509.06262 [cs.FL]. [Online]. Available: https://arxiv.org/abs/2509.06262

[45] H. B. Hunt, D. J. Rosenkrantz, and T. G. Szymanski, "On the equivalence, containment, and covering problems for the regular and context-free languages," *Journal of Computer and System Sciences*, vol. 12, no. 2, pp. 222–268, 1976.

[46] R. E. Stearns and H. B. Hunt III, "On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata," *SIAM Journal on Computing*, vol. 14, no. 3, pp. 598–611, 1985.

[47] F. Henglein and L. Nielsen, "Regular expression containment: coinductive axiomatization and computational interpretation," in *POPL 2011*. ACM, 2011, pp. 385–398.

[48] V. M. Gluskov, "Abstract theory of automata," *Russian Mathematical Surveys*, no. 5, pp. 1–53, 1961.

[49] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[50] M. Tappler, B. K. Aichernig, and F. Lorber, "Timed automata learning via SMT solving," in *Proceedings of the 14th International Symposium on NASA Formal Methods, NFM 2022*, ser. LNCS, vol. 13260. Springer, 2022, pp. 489–507.

[51] B. Barbot, N. Basset, and A. Donzé, "Wordgen : a timed word generation tool," in *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2023*. ACM, 2023, pp. 16:1–16:7.