2017-08-29

Time window temporal logic

Time Window Temporal Logic

Cristian-Ioan Vasile
Division of Systems Engineering
Boston University
Brookline, Massachusetts 02446
Email: cvasile@bu.edu

Derya Aksaray
Department of Mechanical Engineering
Boston University
Boston, Massachusetts 02215
Email: daksaray@bu.edu

Calin Belta
Department of Mechanical Engineering
Boston University
Boston, Massachusetts 02215
Email: cbelta@bu.edu

***Abstract*—This paper introduces *time window temporal logic* (TWTL), a rich expressivity language for describing various time bounded specifications. In particular, the syntax and semantics of TWTL enable the compact representation of serial tasks, which are typically seen in robotics and control applications. This paper also discusses the relaxation of TWTL formulae with respect to deadlines of tasks. Efficient automata-based frameworks to solve synthesis, verification and learning problems are also presented. The key ingredient to the presented solution is an algorithm to translate a TWTL formula to an annotated finite state automaton that encodes all possible temporal relaxations of the specification. Case studies illustrating the expressivity of the logic and the proposed algorithms are included.**

I. Introduction

Temporal logic provides a mathematical formalism to reason about (concurrent) events in terms of time. Due to its rich expressivity, it has been widely used as a specification language to describe properties related to correctness, termination, mutual exclusion, reachability, or liveness [27]. Recently, there has been a great interest in using temporal logic formulae in the analysis and control of dynamical systems for robotic applications. For example, linear temporal logic (LTL) [4] has been extensively used in motion planning and control of robotic systems, e.g., [35, 17, 1, 38, 7, 5, 37, 18, 10, 20, 23].

In robotics applications, the tasks may involve some time constraints (e.g., [31, 29]). For example,

- every visit to A needs to be immediately followed by servicing B within 5 time units;
- two consecutive visits to A need to be at least 10 time units apart;
- visiting A and servicing B need to be completed before the time reaches 15.

Such tasks cannot be described by LTL formulae since LTL cannot deal with temporal properties with explicit time constraints. Therefore, bounded temporal logics are used to capture the time constraints over the tasks. Some examples are bounded linear temporal logic (BLTL) [32, 16], metric temporal logic (MTL) [19], or signal temporal logic (STL) [26].

In this paper, we propose a new specification language called *time window temporal logic* (TWTL). The semantics of TWTL is rich enough to express a wide variety of time-bounded specifications, e.g., "Service A for 3 time units within the time interval $[0, 5]$ and after that service B for 2 time units within $[4, 9]$. If C is serviced for 2 time units within 9 time units, then D should be serviced for 3 time units within the same time interval (i.e., within 9 time units). For instance, some multi-robot persistent surveillance specifications are expressed as TWTL formulae in [36] and [2]. Moreover, we define the notion called *temporal relaxation* of a TWTL formula, which is a quantity computed over the time intervals of a given TWTL formula. In this respect, if the temporal relaxation is

- *negative*, then the tasks expressed in the TWTL formula should be completed before their designated time deadlines, thus satisfying the relaxed formula implies the satisfaction of temporally more strict TWTL formula;
- *zero*, then the relaxed formula is exactly same as the original TWTL formula;
- *positive*, then some tasks expressed in the TWTL formula are allowed to be completed after their designated time deadlines, thus satisfying the relaxed formula implies the violation of the original TWTL formula (or the satisfaction of temporally less strict formula).

We also present an automata-based framework for minimizing the temporal relaxation of a given TWTL formula in problems related to verification, synthesis, and learning. In the theoretical computer science literature, finite languages and the complexity of construction their corresponding automata have been extensively studied [25, 14, 6, 11, 8]. The algorithms proposed in this paper are specialized to handle TWTL formulae and produce the annotated automata, which is used to solve synthesis, verification and learning problems efficiently.

The proposed language TWTL has several advantages over the existing temporal logics. First, a desired specification can be represented in a more compact and comprehensible way in TWTL than BLTL, MTL, or STL. For example, any deadlines expressed in a TWTL formula indicates the exact time bounds as opposed to an STL formula where the time bounds can be shifted. Consider a specification as "stay at A for 4 time steps within the time window $[0, 10]$", which can be expressed in TWTL as $[H^4 A]^{[0,10]}$. The same specification can be expressed in STL as $F_{[0,10-4]} G_{[0,4]} A$ where the outermost time window needs to be modified with respect to the inner time window. Furthermore, compared to BLTL and MTL, the existence of explicit concatenation operator results in a compact representation for serial tasks that are prevalent in robotics and control applications. Under some mild assumptions, we provide a very efficient (linear-

time) algorithm to handle concatenation of tasks. This is in contrast to the general result from computer science that concatenation of languages, even finite ones [25], is exponential in the worst case. Second, the notion of temporal relaxation enables a generic framework to construct the automaton of all possible relaxations of a TWTL formula. In literature, there are some studies investigating the control synthesis problems for minimal violations of LTL fragments [30, 34, 33, 24, 12]. However, the special automaton proposed in this paper is a compact representation of all possible relaxations, which can be used in a variety of problems related to synthesis, verification, or learning to achieve minimal relaxations. Third, for a given TWTL formula, the complexity of constructing automata is independent of the corresponding time bounds. To achieve this property, we exploit the structure of finite languages encoded by TWTL formulae.

The main contributions of this paper are: 1) introducing a new specification language called TWTL, 2) defining *temporal relaxation* of a TWTL formula, 3) presenting a set of provably-correct algorithms to construct the automaton of a given TWTL formula (both for the relaxed and unrelaxed cases), 4) formulating a generic problem in terms of temporal relaxation of a TWTL formula, which can also be specialized into various problems such as verification, synthesis, or learning, and 5) developing a Python package to solve the three specialized problems.

II. Preliminaries

In this section, we introduce the notation and briefly review the main concepts from formal languages, automata theory, and formal verification. For a detailed exposition of these topics, the reader is refereed to [4, 15] and the references therein.

Given $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^n$, $n \geq 2$, the relationship $\mathbf{x} \sim \mathbf{x}'$, where $\sim \in \{<, \leq, >, \geq\}$, is true if it holds pairwise for all components. $\mathbf{x} \sim a$ denotes $\mathbf{x} \sim a\mathbf{1}_n$, where $a \in \mathbb{R}$ and $\mathbf{1}_n$ is the n-dimensional vector of all ones. The extended set of real numbers is denoted by $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$

Let Σ be a finite set. We denote the cardinality and the power set of Σ by $|\Sigma|$ and 2^Σ, respectively. A *word* over Σ is a finite or infinite sequence of elements from Σ. In this context, Σ is also called an *alphabet*. The length of a word w is denoted by $|w|$ (e.g., $|w| = \infty$ if w is an infinite word). Let k, $i \leq j$ be non-negative integers. The k-th element of w is denoted by w_k, and the sub-word $w_i, \ldots, w_j$ is denoted by $w_{i,j}$. A set of words over an alphabet Σ is called a *language* over Σ. The languages of all finite and infinite words over Σ are denoted by Σ^* and Σ^ω, respectively.

Definition II.1 (Prefix language)**.** *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages. We say that $\mathcal{L}_1$ is a prefix language of $\mathcal{L}_2$ if and only if every word in $\mathcal{L}_1$ is a prefix of some word in $\mathcal{L}_2$, i.e., for each word $w \in \mathcal{L}_1$ there exists $w' \in \mathcal{L}_2$ such that $w = w'_{0,i}$, where $0 \leq i < |w'|$, The maximal prefix language of a language $\mathcal{L}$ is denoted by $P(\mathcal{L}) = \{w_{0,i} \mid w \in \mathcal{L}, i \in \{0, \ldots, |w| - 1\}\}$.*

Definition II.2 (Unambiguous language)**.** *A language $\mathcal{L}$ is called unambiguous language if no proper subset L of $\mathcal{L}$ is a prefix language of $\mathcal{L} \setminus L$.*

The above definition immediately implies that a word in an unambiguous language can not be the prefix of another word. Moreover, it is easy to show that the converse is also true.

Definition II.3 (Language concatenation)**.** *Let $\mathcal{L}_1$ be a language over finite words, and let $\mathcal{L}_2$ be a language over finite or infinite words. The concatenation language $\mathcal{L}_1 \cdot \mathcal{L}_2$ is defined as the set of all words ww', where $w \in \mathcal{L}_1$ and $w' \in \mathcal{L}_2$.*

Definition II.4 (Deterministic Finite State Automaton)**.** *A deterministic finite state automaton (DFA) is a tuple $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta_\mathcal{A}, F_\mathcal{A})$, where:*

- *$S_\mathcal{A}$ is a finite set of states;*
- *$s_0 \in S_\mathcal{A}$ is the initial state;*
- *Σ is the input alphabet;*
- *$\delta_\mathcal{A} : S_\mathcal{A} \times \Sigma \to S_\mathcal{A}$ is the transition function;*
- *$F_\mathcal{A} \subseteq S_\mathcal{A}$ is the set of accepting states.*

A transition $s' = \delta_\mathcal{A}(s, \sigma)$ is also denoted by $s \xrightarrow{\sigma}_\mathcal{A} s'$. A trajectory of the DFA $\mathbf{s} = s_0 s_1 \ldots s_{n+1}$ is generated by a finite sequence of symbols $\boldsymbol{\sigma} = \sigma_0 \sigma_1 \ldots \sigma_n$ if $s_0 \in S_\mathcal{A}$ is the initial state of $\mathcal{A}$ and $s_k \xrightarrow{\sigma_k}_\mathcal{A} s_{k+1}$ for all $k \geq 0$. The trajectory generated by $\boldsymbol{\sigma}$ is also denoted by $s_0 \xrightarrow{\boldsymbol{\sigma}}_\mathcal{A} s_{n+1}$. A finite input word $\boldsymbol{\sigma}$ over Σ is said to be accepted by a finite state automaton $\mathcal{A}$ if the trajectory of $\mathcal{A}$ generated by $\boldsymbol{\sigma}$ ends in a state belonging to the set of accepting states, i.e., $F_\mathcal{A}$. A DFA is called *blocking* if the $\delta_\mathcal{A}(s, \sigma)$ is a partial function, i.e., the value of the function is not defined for all values in the domain. A blocking automaton rejects words $\boldsymbol{\sigma}$ if there exists $k \geq 0$ such that $s_k \xrightarrow{\sigma_k}_\mathcal{A} s_{k+1}$ is not defined. The *(accepted) language* corresponding to a DFA $\mathcal{A}$ is the set of accepted input words, which we denote by $\mathcal{L}(\mathcal{A})$.

Definition II.5 (Transition System, TS)**.** *A transition system (TS) is a tuple $\mathcal{T} = (X, x_0, \Delta, AP, h)$, where:*

- *X is a finite set of states;*
- *$x_0 \in X$ is the initial state;*
- *$\Delta \subseteq X \times X$ is a set of transitions;*
- *AP is a set of properties (atomic propositions);*
- *$h : X \to 2^\Pi$ is a labeling function.*

We also denote a transition $(x, x') \in \Delta$ by $x \to_\mathcal{T} x'$. A *trajectory* (or run) of the system is an infinite sequence of states $\mathbf{x} = x_0 x_1 \ldots$ such that $x_k \to_\mathcal{T} x_{k+1}$ for all $k \geq 0$. A state trajectory $\mathbf{x}$ generates an *output trajectory* $\mathbf{o} = o_0 o_1 \ldots$, where $o_k = h(x_k)$ for all $k \geq 0$. The *(generated) language* corresponding to a TS $\mathcal{T}$ is the set of all generated output words, which we denote by $\mathcal{L}(\mathcal{T})$.

III. Time Window Temporal Logic

Time window temporal logic (TWTL) was first introduced in [36] as a rich specification language for robotics applications. TWTL formulae are able to capture temporal logic specifications about the service time windows and their durations.

TWTL is a linear-time logic encoding sets of discrete-time sequences with values in a finite alphabet.

A TWTL formula is defined over a set of atomic propositions AP and has the following syntax:

$$\phi ::= H^d s \,|\, H^d \neg s \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2 \,|\, \neg\phi_1 \,|\, \phi_1 \cdot \phi_2 \,|\, [\phi_1]^{[a,b]}$$

where s is either the "true" constant $\top$ or an atomic proposition in AP; $\wedge$, $\vee$, and $\neg$ are the conjunction, disjunction, and negation Boolean operators, respectively; $\cdot$ is the concatenation operator; H^d with $d \in \mathbb{Z}_{\geq 0}$ is the *hold* operator; and $[\ ]^{[a,b]}$ with $0 \leq a \leq b$ is the *within* operator.

The semantics of the operators is defined with respect to the finite subsequences of a (possibly infinite) word $\mathbf{o}$ over 2^{AP}. Let $\mathbf{o}_{t_1,t_2}$ be the subsequence of $\mathbf{o}$, which starts at time $t_1 \geq 0$ and ends at time $t_2 \geq t_1$. The *hold* operator $H^d s$ specifies that $s \in AP$ should be repeated for d time units. The semantics of $H^d \neg s$ is defined similarly, but for d time units only symbols from $AP \setminus \{s\}$ should appear. For convenience, if $d = 0$ we simply write s and $\neg s$ instead of $H^0 s$ and $H^0 \neg s$, respectively. The word $\mathbf{o}_{t_1,t_2}$ satisfies $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, or $\neg\phi$ if $\mathbf{o}_{t_1,t_2}$ satisfies both formulae, at least one formula, or does not satisfy the formula, respectively. The *within* operator $[\phi]^{[a,b]}$ bounds the satisfaction of ϕ to the time window $[a, b]$. The concatenation operator $\phi_1 \cdot \phi_2$ specifies that first ϕ_1 must be satisfied, and then immediately ϕ_2 must be satisfied.

Formally, the semantics of TWTL formulae is defined recursively as follows:

$$\begin{aligned}
&\mathbf{o}_{t_1,t_2} \models H^d s && \text{iff } s \in o_t, \forall t \in \{t_1, \ldots, t_1 + d\} \wedge (t_2 - t_1 \geq d)\\
&\mathbf{o}_{t_1,t_2} \models H^d \neg s && \text{iff } s \notin o_t, \forall t \in \{t_1, \ldots, t_1 + d\} \wedge (t_2 - t_1 \geq d)\\
&\mathbf{o}_{t_1,t_2} \models \phi_1 \wedge \phi_2 && \text{iff } (\mathbf{o}_{t_1,t_2} \models \phi_1) \wedge (\mathbf{o}_{t_1,t_2} \models \phi_2)\\
&\mathbf{o}_{t_1,t_2} \models \phi_1 \vee \phi_2 && \text{iff } (\mathbf{o}_{t_1,t_2} \models \phi_1) \vee (\mathbf{o}_{t_1,t_2} \models \phi_2)\\
&\mathbf{o}_{t_1,t_2} \models \neg\phi && \text{iff } \neg(\mathbf{o}_{t_1,t_2} \models \phi)\\
&\mathbf{o}_{t_1,t_2} \models \phi_1 \cdot \phi_2 && \text{iff } (\exists t = \arg\min_{t_1 \leq t < t_2} \{\mathbf{o}_{t_1,t} \models \phi_1\}) \wedge\\
& && \quad (\mathbf{o}_{t+1,t_2} \models \phi_2)\\
&\mathbf{o}_{t_1,t_2} \models [\phi]^{[a,b]} && \text{iff } \exists t \geq t_1 + a \text{ s.t. } \mathbf{o}_{t,t_1+b} \models \phi \wedge (t_2 - t_1 \geq b)
\end{aligned}$$

A word $\mathbf{o}$ is said to satisfy a formula ϕ if and only if there exists $T \in \{0, \ldots, |\mathbf{o}|\}$ such that $\mathbf{o}_{0,T} \models \phi$.

A TWTL formula ϕ can be verified with respect to a bounded word. Accordingly, we define the *time bound* of ϕ, i.e., $\|\phi\|$, as the maximum time needed to satisfy ϕ, which can be recursively computed as follows:

$$\|\phi\| = \begin{cases} \max(\|\phi_1\|, \|\phi_2\|) & \text{if } \phi \in \{\phi_1 \wedge \phi_2, \phi_1 \vee \phi_2\} \\ \|\phi_1\| & \text{if } \phi = \neg\phi_1 \\ \|\phi_1\| + \|\phi_2\| + 1 & \text{if } \phi = \phi_1 \cdot \phi_2 \\ d & \text{if } \phi \in \{H^d s, H^d \neg s\} \\ b & \text{if } \phi = [\phi_1]^{[a,b]} \end{cases} \quad (1)$$

We denote the language of all words satisfying ϕ by $\mathcal{L}(\phi)$. Note that TWTL formulae are used to specify prefix languages of either Σ^* or Σ^ω, where $\Sigma = 2^{AP}$. Moreover, the number of operators in a TWTL formula ϕ is denoted by $|\phi|$.

Some examples of TWTL formulae for a robot servicing at some regions can be as follows:

- *servicing within a deadline:* "service A for 2 time units before 10",

$$\phi_1 = [H^2 A]^{[0,10]} \text{ and } \|\phi_1\| = 10. \quad (2)$$

- *servicing within time windows:* "service A for 4 time units within [3, 8] and B for 2 time units within [4, 7]",

$$\phi_2 = [H^4 A]^{[3,8]} \wedge [H^2 B]^{[4,7]} \text{ and } \|\phi_2\| = 8. \quad (3)$$

- *servicing in sequence:* "service A for 3 time units within [0, 5] and after this service B for 2 time units within [4, 9]",

$$\phi_3 = [H^3 A]^{[0,5]} \cdot [H^2 B]^{[4,9]} \text{ and } \|\phi_3\| = 15. \quad (4)$$

- *enabling conditions:* "if A is serviced for 2 time units within 9 time units, then B should be serviced for 3 time units within the same time interval (i.e., within 9 time units)",

$$\phi_4 = [H^2 A \Rightarrow [H^3 B]^{[2,5]}]^{[0,9]} \text{ and } \|\phi_4\| = 9, \quad (5)$$

where $\Rightarrow$ denotes implication.

TWTL provides some benefits over other time-bounded temporal logics. One of the main benefits of TWTL is the existence of an explicit concatenation operator, which results in compact representation of serial tasks. For instance, the specification in (4) is expressed in TWTL, BLTL, and MTL in Table I, where the MTL formula contains a set of recursively defined sub-formulae connected by disjunctions whereas the BLTL formula contains nested temporal operators with conjunction. In both cases, dealing with the disjunction of numerous sub-formulae or the nested temporal operators with conjunction significantly increases the complexity of constructing the automaton (i.e., in exponential or quadratic way, respectively [25]). On the other hand, stemming from the compact representation of TWTL, we provide a linear-time algorithm to handle the concatenations of tasks under some mild assumptions.

TABLE I: The representation of (4) in TWTL, BLTL, and MTL.

TWTL	$[H^3 A]^{[0,5]} \cdot [H^2 B]^{[4,9]}$
BLTL	$\mathbf{F}^{\leq 5-3}(\mathbf{G}^{\leq 3} A \wedge \mathbf{F}^{\leq 9-2+3} \mathbf{G}^{\leq 2} B)$
MTL	$\bigvee_{i=0}^{5-3}(\mathbf{G}_{[i,i+3]} A \wedge \bigvee_{j=i+3+4}^{i+3+9-2} \mathbf{G}_{[j,j+2]} B)$

In addition to the concatenation operator, the existence of within and hold operators also leads to compact (shorter length) representation of specifications, which greatly improves the readability of the formula. For example, the specification in (3) is expressed in various temporal logics in Table II where the BLTL formula contains nested temporal operators with shifted time windows whereas the MTL formula consists of the disjunction of many sub-formulae.

For automata-based model-checking, a BLTL formula is translated into another off-the-shelf temporal logic (e.g., syntactically co-safe linear temporal logic (scLTL) [21]), for which an existing tool (e.g., *scheck* [22]) for the automaton construction can be used [32]. On the other hand, MTL and STL are very expressive temporal logics that are particularly

TABLE II: The representation of (3) in TWTL, BLTL, and MTL.

TWTL	$[H^4 A]^{[3,8]} \wedge [H^2 B]^{[4,7]}$
BLTL	$\mathbf{F}^{\leq 8-4}\mathbf{G}^{\leq 4} A \wedge \mathbf{F}^{\leq 7-2}\mathbf{G}^{\leq 2} B$
MTL	$\bigvee_{i=3}^{8-4} \mathbf{G}_{[i,i+4]} A \wedge \bigvee_{i=4}^{7-2} \mathbf{G}_{[i,i+2]} B$

used for real-time systems. While there is no finite representation for the satisfying language of STL, timed-automata [3] are used to represent the satisfying language of MTL. Compared to the other temporal logics, TWTL has a significantly lower computational complexity since an automaton for the satisfying language of a TWTL formula can be constructed directly (see Sec. VII) and does not require any clocks to deal with the time constraints (as in timed automata). Finally, for a given TWTL formula ϕ, we show that all possible temporally relaxed ϕ can be encoded to a very compact representation, which is enabled from the definition of temporal relaxation introduced in the next section.

IV. Temporal Relaxation

In this section, we introduce a *temporal relaxation* of a TWTL formula. This notion is used in Sec. V to formulate an optimization problem over temporal relaxations.

To illustrate the concept of temporal relaxation, consider the following TWTL formula:

$$\phi_1 = [H^1 A]^{[0:2]} \cdot \left[H^3 B \wedge [H^2 C]^{[0:4]}\right]^{[1:8]}. \tag{6}$$

In cases where ϕ_1 cannot be satisfied, one question is: what is the "closest" achievable formula that can be performed? Hence, we investigate relaxed versions of ϕ_1. One way to do this is to relax the deadlines for the time windows, which are captured by the *within* operator. Accordingly, a relaxed version of ϕ_1 can be written as

$$\phi_1(\boldsymbol{\tau}) = [H^1 A]^{[0:(2+\tau_1)]} \cdot [H^3 B \wedge [H^2 C]^{[0:(4+\tau_2)]}]^{[1:(8+\tau_3)]}, \tag{7}$$

where $\boldsymbol{\tau} = (\tau_1, \tau_2, \tau_3) \in \mathbb{Z}^3$. Note that a critical aspect while relaxing the time windows is to preserve the feasibility of the formula. This means that all sub-formulae of ϕ enclosed by the *within* operators must take less time to satisfy than their corresponding time window durations.

Definition IV.1 (Feasible TWTL formula)**.** *A TWTL formula ϕ is called feasible, if the time window corresponding to each* within *operator is greater than the duration of the corresponding enclosed task (expressed via the* hold *operators).*

Remark IV.1. *Consider the formula in Eq.*(7)*. For $\phi_1(\boldsymbol{\tau})$ to be a feasible TWTL formula, the following constraint must hold: (i) $2+\tau_1 \geq 1$; (ii) $4+\tau_2 \geq 2$ and (iii) $7+\tau_3 \geq \max\{3, 4+\tau_2\}$. Note that $\boldsymbol{\tau}$ may be non-positive. In such cases, $\phi_1(\boldsymbol{\tau})$ becomes a stronger specification than ϕ_1, which implies that the subtasks are performed ahead of their actual deadlines.*

Let ϕ be a TWTL formula. Then, a $\tau-$relaxation of ϕ is defined as follows:

Definition IV.2 ($\tau-$Relaxation of ϕ)**.** *Let $\boldsymbol{\tau} \in \mathbb{Z}^m$, where m is the number of* within *operators contained in ϕ. The τ-relaxation of ϕ is a* feasible *TWTL formula $\phi(\boldsymbol{\tau})$, where each subformula of the form $[\phi_i]^{[a_i,b_i]}$ is replaced by $[\phi_i]^{[a_i,b_i+\tau_i]}$.*

Remark IV.2. *For any ϕ, $\phi(\mathbf{0}) = \phi$.*

Definition IV.3 (Temporal Relaxation)**.** *Given ϕ, let $\phi(\boldsymbol{\tau})$ be a feasible relaxed formula. The temporal relaxation of $\phi(\boldsymbol{\tau})$ is defined as $|\boldsymbol{\tau}|_{TR} = \max_j(\tau_j)$.*

Remark IV.3. *If a word $o \models \phi(\boldsymbol{\tau})$ with $|\boldsymbol{\tau}|_{TR} \leq 0$, then $o \models \phi$.*

V. Problem formulation

In this section, first, we propose a generic optimization problem over temporal relaxations of a TWTL formula. Then, we show how this setup can be used to formulate verification, synthesis, and learning problems.

The objective of the following optimization problem is to find a feasible relaxed version of a TWTL formula that optimizes a cost function penalizing the sets of satisfying and unsatisfying words, and the vector of relaxations.

Problem V.1. *Let ϕ be a TWTL formula over the set of atomic propositions AP, and let $\mathcal{L}_1$ and $\mathcal{L}_2$ be any two languages over the alphabet $\Sigma = 2^{AP}$. Consider a cost function $F : \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \times \mathbb{Z}^m \to \overline{\mathbb{R}}$, where m is the number of* within *operators contained in ϕ. Find $\boldsymbol{\tau}$ such that $F(|\mathcal{L}(\phi(\boldsymbol{\tau})) \cap \mathcal{L}_1|, |\mathcal{L}(\neg\phi(\boldsymbol{\tau})) \cap \mathcal{L}_2|, \boldsymbol{\tau})$ is minimized.*

A. Verification, synthesis, and learning

In the following, we use Problem V.1 to formulate three problems for verification, synthesis, and learning.

1) Verification: [1] Given a transition system $\mathcal{T}$ and a TWTL formula ϕ, we want to check if there exists a relaxed formula $\phi(\boldsymbol{\tau})$ such that all output words generated by $\mathcal{T}$ satisfy $\phi(\boldsymbol{\tau})$.

In Problem V.1, we can set $\mathcal{L}_1 = \emptyset$ and $\mathcal{L}_2 = \mathcal{L}(\mathcal{T})$, and we choose the following cost function:

$$F(x, y, \boldsymbol{\tau}) = 1 - \delta(y), \tag{8}$$

where $x, y \in \mathbb{Z}_{\geq 0}$ and $\delta(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0 \end{cases}$. The cost function in Eq. (8) has a single global minimum value at 0 which corresponds to the case $\mathcal{L}(\mathcal{T}) \cap \mathcal{L}(\neg\phi(\boldsymbol{\tau})) = \emptyset$.

2) Synthesis: Given a transition system $\mathcal{T}$ and a TWTL formula ϕ, we want to find a policy (a trajectory of $\mathcal{T}$) that produces an output word satisfying a relaxed version $\phi(\boldsymbol{\tau})$ of the specification with minimal temporal relaxation $|\boldsymbol{\tau}|_{TR}$.

In Problem V.1, we can set $\mathcal{L}_1 = \mathcal{L}(\mathcal{T})$ and $\mathcal{L}_2 = \emptyset$, and we choose the following cost function:

$$F(x, y, \boldsymbol{\tau}) = \begin{cases} |\boldsymbol{\tau}|_{TR} & x > 0 \\ \infty & \text{otherwise} \end{cases}, \tag{9}$$

[1] This problem is not a verification problem in the usual sense, but rather finding a formula that is satisfied by all runs of a system.

where $x, y \in \mathbb{Z}_{\geq 0}$. The cost function in Eq. (9) is minimized by an output word of $\mathcal{T}$, which satisfies the relaxed version of ϕ with minimum temporal relaxation, see Def. IV.3.

3) Learning: Let ϕ be a TWTL formula and $\mathcal{L}_p$ and $\mathcal{L}_n$ be two finite sets of words labeled as positive and negative examples, respectively. We want to find a relaxed formula $\phi(\boldsymbol{\tau})$ such that the misclassification rate, i.e., $|\{w \in \mathcal{L}_p \mid w \not\models \phi(\boldsymbol{\tau})\}| + |\{w \in \mathcal{L}_n \mid w \models \phi(\boldsymbol{\tau})\}|$, is minimized.

This case can be mapped to the generic formulation by setting $\mathcal{L}_1 = \mathcal{L}_n$, $\mathcal{L}_2 = \mathcal{L}_p$ and choosing the cost function

$$F(x, y, \boldsymbol{\tau}) = x + y, \tag{10}$$

which captures the misclassification rate, where $x, y \in \mathbb{Z}_{\geq 0}$.

B. Overview of the solution

We propose an automata-based approach to solve the verification, synthesis, and learning problems defined above. Specifically, the proposed algorithm constructs an annotated DFA $\mathcal{A}_\infty$, which captures all temporal relaxations of the given formula ϕ, i.e., $\mathcal{L}(\mathcal{A}_\infty) = \mathcal{L}(\phi(\infty))$ (see Def. VI.4 for the definition of $\phi(\infty)$). Note that the algorithm can also be used to construct a (normal) DFA $\mathcal{A}$ which accepts the satisfying language of ϕ, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\phi)$. Using the resulting DFA $\mathcal{A}_\infty$, we proceed in Sec. VIII to solve the synthesis and verification problems using a product automaton approach. For the synthesis problem, we propose a recursive algorithm that computes a satisfying path with minimum temporal relaxation. The learning problem is solved by inferring the minimum relaxation for each trajectory and then combining these relaxations to ensure minimum misclassification rate.

VI. Properties of TWTL

In this section, we present properties of TWTL formulae, their temporal relaxations, and their accepted languages.

In this paper, languages are represented in three ways: as TWTL formulae, as automata, and as sets. As one might expect, there is a duality between some operators of TWTL and set operations, i.e., conjunction, disjunction, and concatenation correspond to intersection, union, and concatenation languages, respectively. Negation may be mapped to complementation with respect to the language of all bounded words, where the bound is given by the time bound of the negated formula.

Proposition VI.1. *The following properties hold*

$$(\phi_1 \cdot \phi_2) \cdot \phi_3 = \phi_1 \cdot (\phi_2 \cdot \phi_3) \tag{11}$$
$$\phi_1 \cdot (\phi_2 \vee \phi_3) = (\phi_1 \cdot \phi_2) \vee (\phi_1 \cdot \phi_3) \tag{12}$$
$$[\phi_1 \vee \phi_2]^{[a,b]} = [\phi_1]^{[a,b]} \vee [\phi_2]^{[a,b]} \tag{13}$$
$$\neg(H^d p) = [\neg p]^{[0,d]} \tag{14}$$
$$[\phi_1]^{[a_1,b_1]} = (H^{a_1-1}\top) \cdot [\phi_1]^{[0,b_1-a_1]} \tag{15}$$
$$(H^{d_1} p) \cdot (H^{d_2} p) = H^{d_1+d_2+1} p \tag{16}$$
$$[\phi_1]^{[a,b]} \Rightarrow [\phi_1]^{[a,b+\tau]} \tag{17}$$
$$(\phi_1 \Rightarrow \phi_2) \Rightarrow ([\phi_1]^{[a,b]} \Rightarrow [\phi_2]^{[a,b]}) \tag{18}$$

where ϕ_1, ϕ_2, and ϕ_3 are TWTL formulae, $p \in \{s, \neg s\}$, $s \in AP \cup \{\top\}$, and $a, b, a_1, b_1, d, d_1, d_2, \tau \in \mathbb{Z}_{\geq 0}$ such that $a \leq b$ and $1 \leq a_1 \leq b_1$.

Proof: These follow directly from the semantics of TWTL formulae. ■

Definition VI.1 (Disjunction-Free Within form)**.** *Let ϕ be a TWTL formula. We say that ϕ is in Disjunction-Free Within (DFW) form if for all* within *operators contained in the formula the associated enclosed subformulae do not contain any disjunction operators.*

An example of a TWTL formula in DFW form is $\phi_1 = [H^2 A]^{[0,9]} \vee [H^5 B]^{[0,9]}$, while a formula not in DFW form is $\phi_2 = [H^2 A \vee H^5 B]^{[0,9]}$. However, ϕ_1 and ϕ_2 are equivalent by Eq. (13) of Prop. VI.1. The next proposition formalizes this property.

Proposition VI.2. *For any TWTL formula ϕ, if the negation operators are only in front of the atomic propositions, then ϕ can be written in the DFW form.*

Proof: The result follows from the properties of distributivity of Boolean operators and Prop. VI.1, which can be applied iteratively to move all disjunction operators outside the *within* operators. ■

In the following, we define the notion of unambiguous concatenation, which enables tracking of progress for sequential specifications. Specifically, if the property holds, then an algorithm is able to decide at each moment if the first specification has finished while monitoring the satisfaction of two sequential specifications.

Definition VI.2. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages. We say that the language $\mathcal{L}_1 \cdot \mathcal{L}_2$ is an* unambiguous concatenation *if each word in the resulting language can be split unambiguously, i.e., $\left(L_1, \mathcal{L}_1, \mathcal{L}_1 \cdot (P(\mathcal{L}_2) \setminus \{\epsilon\})\right)$ is a partition of $P(\mathcal{L}_1 \cdot \mathcal{L}_2)$, where $L_1 = \{w_{0,i} \mid w \in \mathcal{L}_1, i \in \{0, \ldots, |w| - 2\}\}$ and $P(L)$ denotes the maximal prefix language of L.*

The three sets of the partition from Def. VI.2 may be thought as indicating whether the first specification is in progress, the first specification has finished, and the second specification is in progress, respectively.

Proposition VI.3. *Consider two languages $\mathcal{L}_1$ and $\mathcal{L}_2$. The language $\mathcal{L}_1 \cdot \mathcal{L}_2$ is an unambiguous concatenation if and only if $\mathcal{L}_1$ is an unambiguous language.*

Proof: See App. XII-A ■

In the following results, we frequently use the notion of abstract syntax tree of a TWTL formula.

Definition VI.3. *An* Abstract Syntax Tree (AST) *of ϕ is denoted by $AST(\phi)$, where each leaf corresponds to a* hold *operator and each intermediate node corresponds to a Boolean, concatenation, or* within *operator.*

Given a TWTL formula ϕ, there might exist multiple AST trees that represent ϕ. In this paper, $AST(\phi)$ is assumed to

be computed by an LL(*) parser [28]. The reader is referred to [15] for more details on AST and parsers. An example of an AST tree of Eq. (6) is illustrated in Fig. 1.

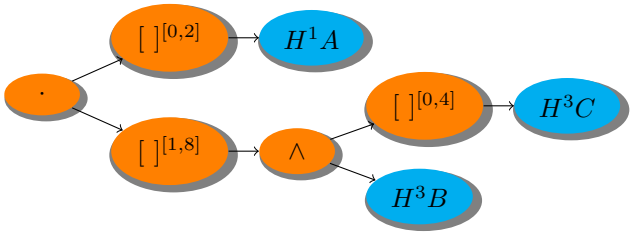


Fig. 1: An AST corresponding to the TWTL in Eq. (6). The intermediate orange nodes correspond to the Boolean, concatenation, and *within* operators, while the cyan leaf nodes represent the *hold* operators.

Proposition VI.4. *Let $\boldsymbol{\tau}', \boldsymbol{\tau}'' \in \mathbb{Z}^m$ such that $\phi(\boldsymbol{\tau}')$ and $\phi(\boldsymbol{\tau}'')$ are two feasible relaxed formulae, where m is the number of* within *operators in ϕ. If $\boldsymbol{\tau}' \leq \boldsymbol{\tau}''$, then $\phi(\boldsymbol{\tau}') \Rightarrow \phi(\boldsymbol{\tau}'')$.*

Proof: See App. XII-B ■

Definition VI.4. *Given an output word $\mathbf{o}$, we say that $\mathbf{o}$ satisfies $\phi(\infty)$, i.e., $\mathbf{o} \models \phi(\infty)$, if and only if $\exists \boldsymbol{\tau}' < \infty$ s.t. $\mathbf{o} \models \phi(\boldsymbol{\tau}')$.*

The next corollary follows directly from Prop. VI.4.

Corollary VI.5. *Let $\boldsymbol{\tau} < \infty$, then $\phi(\boldsymbol{\tau}) \Rightarrow \phi(\infty)$, $\forall \boldsymbol{\tau}$.*

Proposition VI.6. *Let $\phi(\boldsymbol{\tau}')$ and $\phi(\boldsymbol{\tau}'')$ be two feasible relaxed formulae. If $\boldsymbol{\tau}' \leq \boldsymbol{\tau}''$, then $\|\phi(\boldsymbol{\tau}')\| \leq \|\phi(\boldsymbol{\tau}'')\|$.*

Proof: The result follows by structural induction from Eq. (1) using a similar argument as in the proof of Prop. VI.4, see App. XII-B ■

An important observation about TWTL is that the accepted languages corresponding to formulae are finite languages. In the following, we characterize such languages in terms of the associated automata.

Definition VI.5. *A DFA is called* strict *if and only if (i) the DFA is blocking, (ii) all states reach a final state, and (ii) all states are reachable from the initial state.*

Proposition VI.7. *Any DFA $\mathcal{A}$ may be converted to a strict DFA in $O(|S_\mathcal{A}|)$ time.*

Proof: States unreachable from the initial state can be identified by traversing the automaton graph from the initial state using either breath- or depth-first search. Similarly, the states not reaching a final state can be removed by traversing the automaton graph using the reverse direction of the transitions. Both operations take at most $O(|\delta_\mathcal{A}|) = O(|S_\mathcal{A}|)$, since there are at most $|\Sigma|$ transitions outgoing from each state, where Σ is the alphabet of $\mathcal{A}$. ■

Note that a strict DFA is not necessarily minimal with respect to the number of states.

Proposition VI.8. *If $\mathcal{L}$ is a finite language over an alphabet Σ, then the corresponding strict DFA is a directed acyclic graph (DAG). Moreover, given a (general) DFA $\mathcal{A}$, checking if its associated language $\mathcal{L}(\mathcal{A})$ is finite takes $O(|S_\mathcal{A}|)$ time.*

Proof: For the first part, assume for the sake of contradiction that $\mathcal{A}$ has a cycle. Then, we can form words in the accepted language by traversing the cycle $n \in \mathbb{Z}_{\geq 0}$ times before going to a final state. Note that the states in the cycle are reachable from the initial state and also reach a final state, because $\mathcal{A}$ is a strict DFA. It follows that $\mathcal{L}$ is infinite, which contradicts the hypothesis. Checking if a DFA $\mathcal{A}$ is DAG takes $O(|S_\mathcal{A}|)$ by using a topological sorting algorithm, because of the same argument as in Prop. VI.7. ■

Corollary VI.9. *Let $\mathcal{L}$ be a finite unambiguous language over the alphabet Σ and $\mathcal{A}$ be its corresponding strict DFA. The following two statements hold:*

1) *if $s \in F_\mathcal{A}$, then the set of outgoing transitions of s is empty.*
2) *$\mathcal{A}$ may be converted to a DFA with only one final states.*

Proof: Consider a final state $s \in F_\mathcal{A}$. Assume that there exists $s' \in S_\mathcal{A}$ such that $s \xrightarrow{\sigma}_\mathcal{A} s'$, where $\sigma \in \Sigma$. Since $\mathcal{A}$ is strict, it follows that there is another final state $s'' \in F_\mathcal{A}$ which can be reached from s'. Next, we form the words w and w' leading to s and s'' passing trough s', respectively. Clearly, w is a prefix of w', which implies that $\mathcal{L}$ is not an unambiguous language. The second statement follows from the first by noting that in this case, merging all final states does not change the accepted language of the DFA $\mathcal{A}$. ■

VII. Automata construction

In this section, we present a recursive procedure to construct DFAs for TWTL formulae and their temporal relaxations. The resulting DFA are used in Sec. VIII to solve the proposed problems in Sec. V-A.

Throughout the paper, a TWTL formula is assumed to have the following properties:

Assumption 1. *Let ϕ be a TWTL. Assume that (i) negation operators are only in front of atomic propositions, and (ii) all sub-formulae of ϕ correspond to unambiguous languages.*

The second part (ii) of Assump. 1 is a desired property of specifications in practice, because it is related to the tracking of progress towards the satisfaction of the tasks. More specifically, if (ii) holds, then the end of each sub-formula can be determined unambiguously, i.e., without any look-ahead.

A. Construction Algorithm

In [36], a TWTL formula ϕ is translated to an equivalent scLTL formula, and then an off-the-shelf tool, such as *scheck* [22] and *spot* [9], is used to obtain the corresponding DFA. In this paper, we propose an alternative construction, shown in Alg. 1, with two main advantages: (i) the proposed algorithm is optimized for TWTL formulae so it is significantly faster than the method used in [36], and (ii)

the same algorithm can be used to construct a special DFA, which captures all τ-relaxations of ϕ, i.e., the DFA $\mathcal{A}_\infty$ corresponding to $\phi(\infty)$.

Algorithm 1: Translation algorithm – $translate(\cdot)$

Input: ϕ – the specification as a TWTL formula in DFW form
Output: $\mathcal{A}$ – translated DFA

1 **if** $\phi = \phi_1 \otimes \phi_2$, *where* $\otimes \in \{\wedge, \vee, \cdot\}$ **then**
2 $\quad \mathcal{A}_1 \leftarrow translate(\phi_1)$, $\mathcal{A}_2 \leftarrow translate(\phi_2)$
3 $\quad \mathcal{A} \leftarrow \varrho_\otimes(\mathcal{A}_1, \mathcal{A}_2)$
4 **else if** $\phi = H^d p$, *where* $p \in \{s, \neg s\}$ *and* $s \in AP$ **then**
5 $\quad \mathcal{A} \leftarrow \varrho_H(p, d, AP)$
6 **else if** $\phi = [\phi_1]^{[a,b]}$ **then**
7 $\quad \mathcal{A}_1 \leftarrow translate(\phi_1)$
8 $\quad$ **if** *inf* **then** $\mathcal{A} \leftarrow \varrho_\infty(\mathcal{A}_1, a, b)$
9 $\quad$ **else** $\mathcal{A} \leftarrow \varrho_{[]}(\mathcal{A}_1, a, b)$
10 **return** $\mathcal{A}$

Alg. 1 constructs the DFA recursively by traversing $AST(\phi)$ computed via an LL(*) parser [15, 28] from the leaves to the root. If the parameter *inf* is true, then the returned DFA is an annotated DFA $\mathcal{A}_\infty$ corresponding to $\phi(\infty)$; otherwise a normal DFA $\mathcal{A}$ is returned. Each operator has an associated algorithm $\varrho_\otimes$ with $\otimes \in \{\wedge, \vee, \cdot, H, \infty, [\,]\}$, which takes the DFAs corresponding to the operands (subtrees of the operator node in the AST) as input. Then, $\varrho_\otimes$ returns the DFA that accepts the formula associated with the operator node. In the following, we present elaborate on all operators and related operations, such as annotating a DFA, relabeling the states of a DFA, or returning the truncated version of a DFA with respect to some given bound.

B. Annotation

The algorithms presented in this section use DFAs with some additional annotation. In this subsection, we introduce a annotated DFA and two algorithms, Alg. 3 and Alg. 2, that are used to (re)label DFAs and the associated annotation data, respectively.

We assume the following conventions to simplify the notation: (i) there is a global boolean variable *inf* accessible by all algorithms, which specifies whether the normal or the annotated DFAs are to be computed; (ii) in all algorithms, we have $\Sigma = 2^{AP}$; (iii) an element of $\sigma \in \Sigma$ is called a *symbol* and is also a set of atomic propositions, $\sigma \subseteq AP$; (iv) a symbol σ is called *blocking* for a state s if there is no outgoing transition from s activated by σ.

1) Annotation: An *annotated DFA* is a tuple $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, F_\mathcal{A}, T_\mathcal{A})$, where the first five components have the same meaning as in Def. II.4 and $T_\mathcal{A}$ is a tree that corresponds to the AST of the formula associated with the DFA. Each node T of the tree contains the following information:

1) $T.op$ is the operation corresponding to T;
2) $T.I$ is the set of initial states of the automaton corresponding to T;
3) $T.F$ is the set of final states of the automaton corresponding to T;
4) $T.left$ and $T.right$ are the left and right child nodes of T, respectively.

Additionally, if $T.op$ is $\vee$ (disjunction), then T has another attribute $T.choice$, which is explained in Sec. VII-C2.

Note that the associated trees are set to $\emptyset$ and are ignored, if the normal DFAs are computed, i.e., *inf* is false.

The labels of the states change during the construction of the automata. Alg. 2 is used to update the labels stored in the data structures of the tree. The algorithm takes the tree T as input, a mapping m from the states to the new labels, and a boolean value e that specifies if the states are mapped to multiple new states. The first step is to convert the states' new labels to singleton sets if e is false (line 1). Then, the algorithm proceeds to process the tree recursively starting with T. The mapping m is then used to compute $t.I$ and $t.F$ by expanding each state to a set and then computing the union of the corresponding sets (lines 5-6). In the case of $op = \vee$, the three sets B, L, and R, which form the tuple $t.choices$ are also processed. The elements of all three sets are pairs of a state s and a symbol $\sigma \in \Sigma$. Alg. 2 converts the states of all these pairs in the tree sets (lines 7-12).

Algorithm 2: $relabelTree(T, m, e)$

Input: T – a tree structure
Input: m – (complete) relabeling mapping
Input: e – boolean, true if m maps states to sets of states

1 **if** $\neg e$ **then** $m(s) \leftarrow \{m(s)\}, \forall s$
2 $stack \leftarrow [T]$
3 **while** $stack \neq [\,]$ **do**
4 $\quad t \leftarrow stack.pop()$
5 $\quad t.I \leftarrow \bigcup_{s \in t.I} m(s)$
6 $\quad t.F \leftarrow \bigcup_{s \in t.F} m(s)$
7 $\quad$ **if** $op = \vee$ **then**
8 $\quad\quad B, L, R \leftarrow t.choices$
9 $\quad\quad B' \leftarrow \bigcup_{(s_B, \sigma) \in B} \{(s, \sigma) \mid s \in m(s_B)\}$
10 $\quad\quad L' \leftarrow \bigcup_{(s_L, \sigma) \in L} \{(s, \sigma) \mid s \in m(s_L)\}$
11 $\quad\quad R' \leftarrow \bigcup_{(s_R, \sigma) \in R} \{(s, \sigma) \mid s \in m(s_R)\}$
12 $\quad\quad t.choices \leftarrow (B', L', R')$
13 $\quad$ **if** $t.left \neq \emptyset$ **then** $stack.push(t.left)$
14 $\quad$ **if** $t.right \neq \emptyset$ **then** $stack.push(t.right)$

2) Relabeling a DFA: The Alg. 3 relabels the states of a DFA $\mathcal{A}$ with labels given by the mapping m. The map m can be a partial function of the states. The states not specified are labeled with integers starting from i_0 in ascending order. If m is empty, then all states are relabeled with integers. Lastly, if *inf* is true then the tree $T_\mathcal{A}$ associated with the DFA is also relabeled, otherwise it is set as empty.

Algorithm 3: $relabel(\mathcal{A}, m, i_0)$

Input: $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, F_\mathcal{A})$ – a DFA
Input: m – (partial) relabeling mapping
Input: i_0 – start labeling index
Output: the relabeled DFA

1 **for** $s \in S_\mathcal{A}$ *s.t.* $\nexists m(s)$ **do**
2 $m(s) \leftarrow i_0$
3 $i_0 \leftarrow i_0 + 1$
4 $S'_\mathcal{A} \leftarrow \{m(s) \mid s \in S_\mathcal{A}\}$
5 $\delta' \leftarrow \{m(s) \xrightarrow{\sigma}_\mathcal{A} m(s') \mid s \xrightarrow{\sigma}_\mathcal{A} s'\}$
6 $F'_\mathcal{A} \leftarrow \{m(s) \mid s \in F_\mathcal{A}\}$
7 **if** *inf* **then** $T'_\mathcal{A} \leftarrow relabelTree(T_\mathcal{A}, m)$
8 **else** $T'_\mathcal{A} \leftarrow \emptyset$
9 **return** $(S'_\mathcal{A}, m(s_0), \Sigma, \delta', F'_\mathcal{A}, T'_\mathcal{A})$

C. Operators

1) Hold: The DFA corresponding to a *hold* operator is constructed by Alg. 4. The algorithm takes as input an atomic proposition s in positive or negative form, a duration d, and the set of atomic propositions AP. The computed DFA has $d + 2$ states (line 1) that are connected in series as follows: (i) if s is in positive form then the states are connected by all transitions activated by symbols which contain s (lines 2-4); and (ii) if s is in negative form then the states are connected by all transitions activated by symbols which do not contain s (lines 5-7). Lastly, if *inf* is true, a new leaf node is created (line 8).

Algorithm 4: $\varrho_H(p, d, AP)$

Input: $p \in \{s, \neg s\}$, $s \in AP$
Input: d – hold duration
Input: AP – set of atomic propositions
Output: DFA corresponding to $H^d p$

1 $S \leftarrow \{0, \ldots, d+1\}$
2 **if** $p = s$ **then**
3 $\Sigma_s \leftarrow 2^{AP} \setminus 2^{(AP \setminus \{s\})}$
4 $\delta \leftarrow \{i \xrightarrow{\sigma}_\mathcal{A} (i+1) \mid i \in \{0, \ldots, d\}, \sigma \in \Sigma_s\}$
5 **else**
6 $\Sigma_{\neg s} \leftarrow 2^{(AP \setminus \{s\})}$
7 $\delta \leftarrow \{i \xrightarrow{\sigma}_\mathcal{A} (i+1) \mid i \in \{0, \ldots, d\}, \sigma \in \Sigma_{\neg s}\}$
8 **if** *inf* **then** $T \leftarrow tree(H^d, \emptyset, \emptyset, \{0\}, \{d+1\})$
9 **else** $T \leftarrow \emptyset$
10 **return** $(S, 0, 2^{AP}, \delta, \{d+1\}, T)$

2) Conjunction and disjunction: The construction for conjunction and disjunction operations is based on the synchronous product construction and is similar to the standard one [15]. However, $\varrho_\wedge$ and $\varrho_\vee$ produce strict DFAs, which only have one accepting state. Both algorithms recursively construct the product automaton starting from the initial composite state. In the following, we describe the details of the algorithms separately.

Conjunction: The DFA corresponding to the conjunction operation is constructed by Alg. 5. The procedure is recursive and the synchronization condition, i.e., the transition relation, is the following: given two composite states (s_1, s_2) and (s'_1, s'_2), there exists a transition from the first state to the second state if there exists a symbol σ such that: (i) there exists pairwise transitions enabled by σ in the two automata (lines 9-11), i.e., $s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s'_1$ and $s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s'_2$; (ii) one automaton reached a final state and the other has a transition enabled by σ (lines 5-8), i.e., either (a) $s_1 = s'_1 = s_{f1}$ and $s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s'_2$, or (b) $s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s'_1$ and $s_2 = s'_2 = s_{f2}$. The first case covers the synchronous execution (simulation) of both $\mathcal{A}_1$ and $\mathcal{A}_2$ when a symbol is encountered. The second case corresponds to the situation when the two automata require words of different sizes to accept an input. A simple example of this case is the DFA encoding $H^2 A \wedge H^3 B$ and the input word $\{A, B\}, \{A, B\}, \{A, B\}, \{B\}$, which clearly satisfies the TWTL formula.

Algorithm 5: $\varrho_\wedge(\mathcal{A}_1, \mathcal{A}_2)$

Input: $\mathcal{A}_1 = (S_{\mathcal{A}_1}, s_{01}, \Sigma, \delta_1, \{s_{f1}\}, T_{\mathcal{A}_1})$ – left DFA
Input: $\mathcal{A}_2 = (S_{\mathcal{A}_2}, s_{02}, \Sigma, \delta_2, \{s_{f2}\}, T_{\mathcal{A}_2})$ – right DFA
Output: DFA corresponding to $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$

1 $S \leftarrow \{(s_{01}, s_{02})\}$, $E \leftarrow \emptyset$
2 $stack \leftarrow [(s_{01}, s_{02})]$
3 **while** $stack \neq [\,]$ **do**
4 $\boldsymbol{s} = (s_1, s_2) \leftarrow stack.pop()$
5 **if** $s_1 = s_{f1}$ **then**
6 $S_n \leftarrow \{((s_1, s'_2), \sigma) \mid s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s'_2\}$
7 **else if** $s_2 = s_{f2}$ **then**
8 $S_n \leftarrow \{((s'_1, s_2), \sigma) \mid s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s'_1\}$
9 **else**
10 $S_n \leftarrow \{((s'_1, s'_2), \sigma) \mid \exists \sigma \in \Sigma$ s.t.
11 $(s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s'_1) \wedge (s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s'_2)\}$
12 $E \leftarrow E \cup \{(\boldsymbol{s}, \sigma, \boldsymbol{s'}) \mid (\boldsymbol{s'}, \sigma) \in S_n\}$
13 $S' \leftarrow \{\boldsymbol{s'} \mid \exists \sigma \in \Sigma$ s.t. $(\boldsymbol{s'}, \sigma) \in S_n\}$
14 $stack.extends(S' \setminus S)$
15 $S \leftarrow S \cup S'$
16 $m_L = \{(u, \{(u, v) \in S_\mathcal{A}\}) \mid u \in S_{\mathcal{A}_1}\}$
17 $m_R = \{(v, \{(u, v) \in S_\mathcal{A}\}) \mid v \in S_{\mathcal{A}_2}\}$
18 $T_\mathcal{A} \leftarrow tree(\wedge, relabelTree(T_{\mathcal{A}_1}, m_L, \top),$
19 $relabelTree(T_{\mathcal{A}_2}, m_R, \top), \{(s_{01}, s_{02})\}, \{(s_{f1}, s_{f2})\})$
20 $\mathcal{A} \leftarrow (S, (s_{01}, s_{02}), \Sigma, E, \{(s_{f1}, s_{f2})\}, T_\mathcal{A})$
21 **return** $relabel(\mathcal{A}, \emptyset, 0)$

Note that Alg. 5 generates only composite states which are reachable from the initial composite state (s_{01}, s_{02}). The resulting automaton has a single final state (s_{f1}, s_{f2}) which captures the fact that both automata must accept the input word in order for the product automaton to accept it.

After the automaton is constructed, the corresponding tree is created (lines 16-19). The child subtrees are taken from $\mathcal{A}_1$

and $\mathcal{A}_2$, and relabeled. The relabeling mapping expands each state s to the set of all composite states, which have s as the first or second component corresponding to whether s is a state of the left or right automaton, respectively.

Disjunction: The disjunction operations is translated using Alg. 6. The first step of the algorithm is to add a trap state in each of the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (line 1). All states of an automaton, except the final state, are connected via blocking symbols to the trap state $\bowtie$ (lines 3-4). The trap state has self-transitions for all symbols. Afterwards, the algorithm creates the synchronous product automaton in the same way as for the conjunction operation (lines 4-13). However, in this case, we do not need to treat composite states that contain a final state of one of the two automata separately. This follows from the semantics of the disjunction operation, which accepts a word as soon as at least one automaton accepts the word.

Algorithm 6: $\varrho_\vee(\mathcal{A}_1, \mathcal{A}_2)$

Input: $\mathcal{A}_1 = (S_{\mathcal{A}_1}, s_{01}, \Sigma, \delta_1, \{s_{f1}\}, T_{\mathcal{A}_1})$ – left DFA
Input: $\mathcal{A}_2 = (S_{\mathcal{A}_2}, s_{02}, \Sigma, \delta_2, \{s_{f2}\}, T_{\mathcal{A}_2})$ – right DFA
Output: DFA corresponding to $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$

1 $S'_{\mathcal{A}_1} \leftarrow S_{\mathcal{A}_1} \cup \{\bowtie\}$, $S'_{\mathcal{A}_2} \leftarrow S_{\mathcal{A}_2} \cup \{\bowtie\}$
2 $\delta'_1 \leftarrow \delta_1 \cup \{(s, \sigma, \bowtie) \mid s \in S'_{\mathcal{A}_1} \setminus \{s_{f1}\}, \sigma \in \Sigma, \nexists \delta_1(s, \sigma)\}$
3 $\delta'_2 \leftarrow \delta_2 \cup \{(s, \sigma, \bowtie) \mid s \in S'_{\mathcal{A}_2} \setminus \{s_{f2}\}, \sigma \in \Sigma, \nexists \delta_2(s, \sigma)\}$
4 $S \leftarrow \{(s_{01}, s_{02})\}$, $E \leftarrow \emptyset$
5 $stack \leftarrow [(s_{01}, s_{02})]$
6 **while** $stack \neq [\,]$ **do**
7 $\quad \boldsymbol{s} = (s_1, s_2) \leftarrow stack.pop()$
8 $\quad S_n \leftarrow \{((s'_1, s'_2), \sigma) \mid \exists \sigma \in \Sigma$ s.t.
9 $\qquad\qquad (s'_1 = \delta'_1(s_1, \sigma)) \wedge (s'_2 = \delta'_2(s_2, \sigma))\}$
10 $\quad E \leftarrow E \cup \{(\boldsymbol{s}, \sigma, \boldsymbol{s'}) \mid (\boldsymbol{s'}, \sigma) \in S_n\}$
11 $\quad S' \leftarrow \{\boldsymbol{s'} \mid \exists \sigma \in \Sigma$ s.t. $(\boldsymbol{s'}, \sigma) \in S_n\}$
12 $\quad stack.extends(S' \setminus S)$
13 $\quad S \leftarrow S \cup S'$
14 $B \leftarrow \{(\boldsymbol{s}, \sigma) \mid \exists \sigma$ s.t. $(\boldsymbol{s}, \sigma, (s_{f1}, s_{f2})) \in E\}$
15 $L \leftarrow \{(\boldsymbol{s}, \sigma) \mid \exists s_2 \neq s_{f2}, \exists \sigma$ s.t. $(\boldsymbol{s}, \sigma, (s_{f1}, s_2) \in E\}$
16 $R \leftarrow \{(\boldsymbol{s}, \sigma) \mid \exists s_1 \neq s_{f1}, \exists \sigma$ s.t. $(\boldsymbol{s}, \sigma, (s_1, s_{f2}) \in E\}$
17 $F \leftarrow \{(s_1, s_2) \in S \mid (s_1 = s_{f1}) \vee (s_2 = s_{f2})\}$
18 $S \leftarrow S \setminus (F \cup \{(\bowtie, \bowtie)\})$
19 $E \leftarrow E \setminus \{(\boldsymbol{s}, \sigma, \boldsymbol{s'}) \in E \mid \boldsymbol{s'} \in F\}$
20 $E \leftarrow E \cup \{(\boldsymbol{s}, \sigma, (s_{f1}, s_{f2})) \mid (\boldsymbol{s}, \sigma) \in B \cup L \cup R\}$
21 $m_L = \big\{(u, \{(u, v) \in S_{\mathcal{A}}\}) \mid u \in S_{\mathcal{A}_1}\big\}$
22 $m_R = \big\{(v, \{(u, v) \in S_{\mathcal{A}}\}) \mid v \in S_{\mathcal{A}_2}\big\}$
23 $T_{\mathcal{A}} \leftarrow tree(\vee, relabelTree(T_{\mathcal{A}_1}, m_L, \top),$
24 $\quad relabelTree(T_{\mathcal{A}_2}, m_R, \top), \{(s_{01}, s_{02})\}, \{(s_{f1}, s_{f2})\})$
25 $T_{\mathcal{A}}.choices \leftarrow (B, L, R)$
26 $\mathcal{A} \leftarrow (S, (s_{01}, s_{02}), \Sigma, E, \{(s_{f1}, s_{f2})\}, T_{\mathcal{A}})$
27 **return** $relabel(\mathcal{A}, \emptyset, 0)$

In the standard construction [15], the resulting automaton would have multiple final states, which are computed in line 17. However, because finals states do not have outgoing transitions, we can merge all final states and obtain an automaton with only one final state (lines 17-20). The composite trap state is also removed from the set of states (line 18).

The annotation tree is created similarly to the conjunction case (lines 21-24). However, for the disjunction case, we add additional information on the automaton. This information $T.choices$ is used in latter algorithm to determine if a word has satisfied the left, right, or both sub-formulae corresponding to the disjunction formula. This is done by partitioning the transitions incoming into finals states (line 14-16) and storing this partition in the associated tree node (line 25). Note that only the start state and the symbol of each transition is stored in the partition sets and these are well defined, because the DFAs are deterministic.

3) Concatenation: The algorithm to compute an automaton accepting the concatenation language of two languages is shown in Alg. 7. The special structure of the unambiguous languages, see Sec. VI for details, admits a particularly simple and intuitive construction procedure. The composite automaton is obtained by identifying the final state of left automaton $\mathcal{A}_1$ with the initial state of the right automaton $\mathcal{A}_2$.

Algorithm 7: $\varrho_\cdot(\mathcal{A}_1, \mathcal{A}_2)$

Input: $\mathcal{A}_1 = (S_{\mathcal{A}_1}, s_{01}, \Sigma, \delta_1, \{s_{f1}\}, T_{\mathcal{A}_1})$ – left DFA
Input: $\mathcal{A}_2 = (S_{\mathcal{A}_2}, s_{02}, \Sigma, \delta_2, \{s_{f2}\}, T_{\mathcal{A}_2})$ – right DFA
Output: DFA corresponding to $\mathcal{L}(\mathcal{A}_1) \cdot \mathcal{L}(\mathcal{A}_2)$

1 $\mathcal{A}_1 \leftarrow relabel(\mathcal{A}_1, \emptyset, 0)$
2 $\mathcal{A}_2 \leftarrow relabel(\mathcal{A}_2, \{(s_{02}, s_{f1})\}, |S_{\mathcal{A}_1}|)$
3 **if** *inf* **then** $T \leftarrow tree(\cdot, T_{\mathcal{A}_1}, T_{\mathcal{A}_2}, \{s_{01}\}, \{s_{f2}\})$
4 **else** $T \leftarrow \emptyset$
5 **return** $(S_{\mathcal{A}_1} \cup S_{\mathcal{A}_2}, s_{01}, \Sigma, \delta_1 \cup \delta_1, \{s_{f2}\}, T)$

4) Within: There are two algorithms used to construct a DFA associated with a *within* operator, Alg. 8 and Alg. 9 correspond to the relaxed and normal construction (lines 6-9 of Alg. 1).

Relaxed within: The construction procedure Alg. 8 is as follows: starting from the DFA corresponding to the enclosed formula, all states are connected via blocking symbols to the initial state (lines 3-4). The last step is to create a number of a states connected in sequence for all symbols, similarly to Alg. 4, and connecting the a-th state to the initial state also for all symbols (lines 5-8).

Connecting all states to the initial state represents a restart of the automaton in case a blocking symbol was encountered. Thus, the resulting automaton offers infinite retries for a word to satisfy the enclosed formula. The a states added before the initial state represent a delay of length a for the start of the tracking of the satisfaction of the enclosed formula. Note that the procedure and resulting automaton do not depend on the upper bound b.

Normal within: The algorithm for the normal case builds upon Alg. 8. In this case the construction procedure Alg. 9 must take into account the upper time bound b. Similarly to the relaxed case, we need to restart the automaton of the when a blocking symbol is encountered. However, there are two

Algorithm 8: $\varrho_\infty(\mathcal{A}, a, b)$

Input: $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, \{s_f\}, T_\mathcal{A})$ – child DFA
Input: a – lower bound of time-window
Input: b – upper bound of time-window
Output: computed DFA

1 $\mathcal{A} \leftarrow relabel(\mathcal{A}, \emptyset, 0)$
2 $S \leftarrow \emptyset$, $E \leftarrow \emptyset$
3 **for** $s \in S_\mathcal{A} \setminus \{s_f\}$ **do**
4 $\quad E \leftarrow E \cup \{(s, \sigma, s_0) \mid \nexists s' = \delta(s, \sigma)\}$
5 **if** $a > 0$ **then**
6 $\quad S \leftarrow \{|S_\mathcal{A}|, \ldots, |S_\mathcal{A}| + a - 1\}$
7 $\quad E \leftarrow E \cup \big\{(i, \sigma, i+1) \mid i \in S \setminus \{|S_\mathcal{A}| + a - 1\}, \sigma \in \Sigma\big\}$
8 $\quad E \leftarrow E \cup \{(|S_\mathcal{A}| + a - 1, \sigma, s_0) \mid \sigma \in \Sigma\}$
9 $T \leftarrow tree([\,]_\infty^{[a,b]}, T_\mathcal{A}, \emptyset, \{|S_\mathcal{A}|\}, \{s_f\})$
10 **return** $(S_\mathcal{A} \cup S, |S_\mathcal{A}|, \Sigma, \delta \cup E, \{s_f\}, T)$

major differences: (i) the automaton must track the number of restarts, because there are only a finite number of tries depending on the deadline b, and (ii) the automaton $\mathcal{A}$ may need to be truncated for the last restart retries, i.e., all paths must have a length of at most a given length, in order to ensure that the satisfaction is realized before the upper time limit b.

In Alg. 9, first the maximum number of restarts p is computed in lines 1-2. Then, p DFAs are created (lines 3-12), which correspond to the relabeled and truncated copies of $\mathcal{A}$, see Alg. 10, and their union is computed iteratively. The truncation bound is computed as the remaining time units until the limit b is reached. The final state is always labeled with -1 (line 7) and, therefore, the resulting DFA has exactly one final state. Next, the restart transitions are added (lines 13-18). Note that the transitions, enabled by blocking symbols, lead to initial states of the proper restart automaton. For example, if a blocking symbol was encountered after two symbols, then the restart transition (if it exists) leads to the initial state of the fourth copy of the automaton. Lastly, a delay of a time units is added before the initial state of the automaton similar to the relaxed case.

5) Truncate: Alg. 10 takes as input a DFA $\mathcal{A}$ and a cutoff bound l and returns a version of $\mathcal{A}$ with all paths guaranteed to have length at most l. The algorithm is based on a breath-first search and returns a strict DFA.

D. Correctness

The following theorems show that the proposed algorithms for translating TWTL formulae to (normal or annotated) automata are correct.

Theorem VII.1. *If ϕ is a TWTL formula satisfying Assump. 1 and the global parameter inf is true, then Alg. 1 generates a DFA $\mathcal{A}_\infty$ such that $\mathcal{L}(\mathcal{A}_\infty) = \mathcal{L}(\phi(\infty))$.*

Proof: The proof follows by structural induction on $AST(\phi)$ and the properties of TWTL languages.

Algorithm 9: $\varrho_{[\,]}(\mathcal{A}, a, b)$

Input: $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, \{s_f\}, T_\mathcal{A})$ – child DFA
Input: a – lower bound of time-window
Input: b – upper bound of time-window
Output: computed DFA

1 $l \leftarrow Dijkstra(\mathcal{A}, s_0, s_f)$
2 $p \leftarrow b - a - l + 2$
3 $I \leftarrow [\,]$ `// list`
4 $n \leftarrow 0$
5 $\mathcal{A}_r \leftarrow (S_{\mathcal{A}_r} = \emptyset, \infty, \Sigma, \delta_r = \emptyset, \emptyset, \emptyset)$
6 **for** $k \in \{1, \ldots, p\}$ **do**
7 $\quad m \leftarrow \{(s_f, -1)\}$ `// mark final state`
8 $\quad \mathcal{A}_a \leftarrow relabel(\mathcal{A}, m, n)$
9 $\quad \mathcal{A}_t \leftarrow truncate(\mathcal{A}_a, b - a + 2 - k)$
10 $\quad \mathcal{A}_r \leftarrow (S_{\mathcal{A}_r} \cup S_{\mathcal{A}_t}, \infty, \Sigma, \delta_r \cup \delta_t, \{-1\}, \emptyset)$
11 $\quad I \leftarrow I + [s_{0t}]$
12 $\quad n \leftarrow n + |S_{\mathcal{A}_t}|$
13 $S_c \leftarrow \{I[0]\}$, $E \leftarrow \emptyset$
14 **for** $s_r \in I[1:]$ **do**
15 $\quad S_n \leftarrow \emptyset$
16 $\quad$ **for** $s \in S_c \setminus \{-1\}$ **do**
17 $\quad\quad E \leftarrow E \cup \{(s, \sigma, s_r) \mid \sigma \in \Sigma \text{ s.t. } \nexists \delta_r(s, \sigma)\}$
18 $\quad S_c \leftarrow S_c \cup \{s_r\}$
19 $S \leftarrow \emptyset$
20 **if** $a > 0$ **then**
21 $\quad S \leftarrow \{|S_{\mathcal{A}_r}|, \ldots, |S_{\mathcal{A}_r}| + a - 1\}$
22 $\quad E \leftarrow E \cup \big\{(i, \sigma, i+1) \mid i \in S \setminus \{|S_{\mathcal{A}_r}| + a - 1\}, \sigma \in \Sigma\big\}$
23 $\quad E \leftarrow E \cup \{(|S_{\mathcal{A}_r}| + a - 1, \sigma, s_0) \mid \sigma \in \Sigma\}$
24 **return** $(S_{\mathcal{A}_r} \cup S, I[0], \Sigma, \delta_r \cup E, \{-1\}, \emptyset)$

Algorithm 10: $truncate(\mathcal{A}, l)$

Input: $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, \{s_f\}, T_\mathcal{A})$ – a DFA
Input: l – cutoff value
Output: computed DFA

1 $S \leftarrow \{s_0\}$
2 $E \leftarrow \emptyset$
3 $L_n \leftarrow \{s_0\}$
4 **for** $i \in \{1, \ldots, l\}$ **do**
5 $\quad L_c \leftarrow L_n$
6 $\quad L_n \leftarrow \emptyset$
7 $\quad$ **for** $s \in L_c$ **do**
8 $\quad\quad$ **for** $(s_c, \sigma_c) \in \{(s', \sigma) | \exists \sigma \in \Sigma \text{ s.t. } s \xrightarrow{\sigma}_\mathcal{A} s'\}$ **do**
9 $\quad\quad\quad E \leftarrow E \cup (s, \sigma_c, s_c))$
10 $\quad\quad\quad$ **if** $s_c \notin S$ **then**
11 $\quad\quad\quad\quad S \leftarrow S \cup \{s_c\}$
12 $\quad\quad\quad\quad L_n \leftarrow L_n \cup \{s_c\}$
13 $\mathcal{A}_t = (S_\mathcal{A}, s_0, \Sigma, \delta \setminus E, \{s_f\}, T_\mathcal{A})$
14 $S_{traps} = \{s \in S_\mathcal{A} | \nexists \boldsymbol{\sigma} \in \Sigma^* \text{ s.t. } s \xrightarrow{\boldsymbol{\sigma}}_{\mathcal{A}_t} s_f\}$
15 **return** $(S_\mathcal{A} \setminus S_{traps}, s_0, \Sigma, \delta \setminus E, \{s_f\}, T_\mathcal{A})$

Before we proceed with the induction, notice that all construction algorithms associated with the operators of TWTL generate strict DFAs with only one final state without any outgoing transitions.

The base case corresponds to the leaf nodes of $AST(\phi)$ which are associated with *hold* operators, see Fig. 1, and follows by construction from Alg. 4.

The induction hypothesis requires that the theorem holds for the DFAs returned by the recursion in Alg. 1. In the case of the conjunction and disjunction operators, the property follows from the product construction method [15]. The theorem holds also for the concatenation operator, because: (a) the returned DFAs have one final state without any outgoing transitions, and (b) the languages corresponding to the two operand formulae are unambiguous. Thus, the correctness of the construction described in Alg. 7 follows immediately from the unambiguity of the concatenation, see Def. VI.2. Lastly, the case of the *within* operator (relaxed form), follow from the Assump. 1. The *within* operator adds transitions to a DFA from each state to the initial state on all undefined symbols. In other words, the operator restarts the execution of a DFA from the initial state. If there are no disjunction operators, then going back to the initial state is the only correct choice. Otherwise, because of alternative paths induced by disjunction, there might be other states from which the DFA might need to go back to in order to correctly restart. ■

Theorem VII.2. *If ϕ is a TWTL formula satisfying Assump. 1 and the global parameter inf is false, then Alg. 1 generates DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\phi)$.*

Proof: The proof is similar to that of Thm. VII.1 and is omitted for brevity. ■

E. Complexity

In this section, we review the complexity of the algorithms presented in the previous section for the construction of DFAs from TWTL formulae. The complexity of basic composition operations for incomplete and acyclic DFAs has been explored in [25, 14, 6, 11, 8]. Our construction algorithms differ from the ones in the literature because we specialized and optimized them to translate TWTL formulae and handle words over power sets of atomic propositions.

The complexity of the relabeling procedures are $O(|T|)$ and $O(|S_{\mathcal{A}}|)$ corresponding to Alg. 2 and Alg. 3, respectively. The complexity of the *hold* operator Alg. 4 is $O(d \cdot 2^{|AP|})$. The construction algorithms for conjunction and disjunction Alg. 5 and Alg. 6 have the same complexity $O(|S_{\mathcal{A}_1}| \cdot |S_{\mathcal{A}_2}| \cdot 2^{|AP|})$, because these are based on the product automaton construction. Concatenation has complexity $O(|S_{\mathcal{A}_1}| + |S_{\mathcal{A}_2}|)$ due to the relabeling operations. Lastly, the *within* operation can be performed in $O(a \cdot 2^{|AP|} + |S_{\mathcal{A}}| \cdot 2^{|AP|})$ and $O(a \cdot 2^{|AP|} + b\,|S_{\mathcal{A}}| \cdot 2^{|AP|})$ for the infinity Alg. 8 and the normal Alg. 9 construction, respectively, where Alg. 10 used in the normal construction procedure takes $O(|S_{\mathcal{A}}| \cdot 2^{|AP|})$. The overall translation algorithm Alg. 1 takes at most $O(2^{|\phi|+|AP|})$.

It is very important to notice that the infinity construction does not depend on the deadline b, which makes the procedure more efficient than the normal construction.

VIII. Solution

In this section, we will use the following notation. Let T be an annotation tree associated with a DFA. We denote by ϕ_T the TWTL formula corresponding to the tree T. Given a finite sequence $\mathbf{p} = p_0, \ldots, p_n$, we denote the first and the last elements by $b(\mathbf{p}) = p_0$ and $e(\mathbf{p}) = p_n$, respectively.

Definition VIII.1 (Primitive)**.** *Let ϕ be a TWTL formula. We say that ϕ is* primitive *if ϕ does not contain any* within *operators.*

A. Compute temporal relaxation for a word

The automata construction presented in Sec. VII can be used to compute the temporal relaxation of words with repsect to TWTL formulae. Let ϕ be a TWTL formula and $\boldsymbol{\sigma}$ be a word. In this section, we show how to infer (synthesize) a set of temporal relaxations $\boldsymbol{\tau}$ of the deadlines in ϕ such that $\boldsymbol{\sigma}$ satisfies $\phi(\tau)$ and $|\boldsymbol{\tau}|_{TR}$ is minimized. Alg. 11 computes the vector of temporal relaxations corresponding to each *within* operator. First, the annotated DFA $\mathcal{A}_\infty$ is computed together with the associated annotation tree T (line 2). Next, additional annotations are added to the tree T using the $initTreeTR()$ procedure (line 3). Each node corresponding to a *within* operation is assigned three variables $T.ongoing$, $T.done$ and $T.steps$, which track whether the processing of the operator is ongoing, done, and the number of steps to process the operator, respectively. The three variables are initialized to $\bot$, $\bot$, and -1, respectively. Then, Alg. 11 cycles through the symbols of the input word $\boldsymbol{\sigma}$ and updates the tree using $updateTree()$ via Alg. 12. Finally, the temporal relaxation vector is returned by the $evalTreeTR()$ procedure via Alg. 13.

Algorithm 11: $tr(\cdot)$ – Compute temporal relaxation

Input: $\boldsymbol{\sigma}$ a word over the alphabet 2^{AP}
Input: ϕ a TWTL formula
Output: τ^* - minimum maximal temporal relaxation
Output: $\boldsymbol{\tau}$ - temporal relaxation vector

1 **if** ϕ *is primitive* **then return** $(-\infty, [\,])$
2 $\mathcal{A}_\infty, T \leftarrow translate(\phi; \text{inf} = \top)$
3 $initTreeTR(T)$
4 $s_{prev} \leftarrow \bot$; $s_c \leftarrow s_0$
5 $updateTreeTR(T, s_c, s_{prev}, \emptyset, \emptyset)$
6 **for** $\sigma \in \boldsymbol{\sigma}$ **do**
7 **if** $s_c \in F_{\mathcal{A}_\infty}$ **then break**
8 $s_{prev} \leftarrow s_c$
9 $s_c \leftarrow \delta_{\mathcal{A}_\infty}(s_c, \sigma)$
10 $updateTreeTR(T, s_c, s_{prev}, \sigma, \emptyset)$
11 **return** $evalTreeTR(T)$

The tree is updated recursively in Alg. 12. A *within* operator is marked as ongoing, i.e., $T.ongoing = \top$, when the

current state is in the set of initial states associated with the operator (line 2). Similarly, when the current state is in the set of final states associated with the operator, the *within* operator is marked as done (lines 3-6), i.e. $T.done = \top$ and $T.ongoing = \bot$. The number of steps $T.steps$ of all ongoing *within* operators is incremented (line 7).

To enforce correct computation of the temporal relaxation with respect to the disjunction operators, Alg. 12 keeps track of a set of constraints C. The set C is composed of state-symbol pairs, and is used to determine which of the two subformulae of a disjunction are satisfied by the input word (lines 12-17). To achieve this, we use the annotation variables $T.choices$ (see Alg. 6), which capture both cases. For all other operators, the constraint sets are propagated unchanged (lines 8, 10, 11).

Algorithm 12: $updateTreeTR(\cdot)$

Input: s_c – current state
Input: s_{prev} – previous state
Input: σ – current symbol in word
Input: C – set of constraints associated with the states

1 **if** $T.op = [\,]^{[a,b]}$ **then**
2 **if** $s_c \in T.I$ **then** $T.ongoing \leftarrow \top$
3 **if** $s_c \in T.F$ **then**
4 **if** $(C = \emptyset) \vee (\sigma \subseteq C(s_{prev}))$ **then**
5 $T.ongoing \leftarrow \bot$
6 $T.done \leftarrow \top$
7 **if** $T.ongoing$ **then** $T.\tau \leftarrow T.\tau + 1$
8 $updateTreeTR(T.left, s_c, s_{prev}, \sigma, C)$
9 **else**
10 **if** $T.op = \cdot$ **then** $C_L \leftarrow \emptyset$; $C_R \leftarrow C$
11 **else if** $T.op = \wedge$ **then** $C_L \leftarrow C$; $C_R \leftarrow C$
12 **else if** $T.op = \vee$ **then**
13 $C_L \leftarrow T.choices.L \cup T.choices.B$
14 $C_R \leftarrow T.choices.R \cup T.choices.B$
15 **if** $C \neq \emptyset$ **then**
16 $C_L \leftarrow C \cap C_L$
17 $C_R \leftarrow C \cap C_R$
18 $updateTreeTR(T.left, s_c, s_{prev}, \sigma, C_L)$
19 $updateTreeTR(T.right, s_c, s_{prev}, \sigma, C_R)$

Finally, Alg. 13 extracts the temporal relaxation from the annotation tree T after all symbols of the input word $\boldsymbol{\sigma}$ were processed. Alg. 13 also computes the minimum maximum temporal relaxation value, which may be $-\infty$ if ϕ is primitive (line 1). The recursion in Alg. 13 differs between disjunction and the other operators. One subformula is sufficient to hold to satisfy the formula associated with a disjunction operator. Thus, the optimal temporal relaxation is the minimum or maximum between the two optimal temporal relaxations of the subformulae for disjunction (line 12), and conjunction and concatenation (line 13), respectively. Lines 15-16 of Alg. 13 cover the cases involving primitive subformulae.

The complexity of Alg. 11 is $O(2^{|\phi|+|AP|} + |\boldsymbol{\sigma}| \cdot |\phi|)$, where the first term is the complexity of constructing $\mathcal{A}_\infty$ in line 1 and the second term corresponds to the update of the tree for each symbol in $\boldsymbol{\sigma}$ and the final evaluation of the tree.

Algorithm 13: $evalTreeTR(\cdot)$

Input: T – annotated tree
Output: τ^* - minimum maximal temporal relaxation
Output: $\boldsymbol{\tau}$ - temporal relaxation vector

1 **if** ϕ_T *is primitive* **then return** $(-\infty, [\,])$
2 **else if** $T.op = [\phi]^{[a,b]}$ **then**
3 $\tau^*_{ch}, \boldsymbol{\tau}_{ch} = evalTreeTR(tree.left)$
4 **if** $T.done = \top$ **then**
5 **return** $\big(\max\{\tau^*_{ch}, T.steps - b\}, [\boldsymbol{\tau}_{ch}, T.steps - b]\big)$
6 **else**
7 **return** $\big(-\infty, [\boldsymbol{\tau}_{ch}, -\infty]\big)$
8 **else** // $\wedge$, $\vee$ or $\cdot$
9 $\tau^*_L, \boldsymbol{\tau}_L = evalTreeTR(tree.left)$
10 $\tau^*_R, \boldsymbol{\tau}_R = evalTreeTR(tree.right)$
11 **if** $(\tau^*_L \neq -\infty) \wedge (\tau^*_R \neq -\infty)$ **then**
12 **if** $T.op = \vee$ **then** $\tau^* \leftarrow \min\{\tau^*_L, \tau^*_R\}$
13 **else** $\tau^* \leftarrow \max\{\tau^*_L, \tau^*_R\}$
14 **else**
15 **if** $T.op = \vee$ **then** $\tau^* \leftarrow \max\{\tau^*_L, \tau^*_R\}$
16 **else** $\tau^* \leftarrow -\infty$
17 **return** $\big(\tau^*, [\boldsymbol{\tau}_L, \boldsymbol{\tau}_R]\big)$

B. Control policy synthesis for a finite transition system

Let $\mathcal{T}$ be a finite transition system, and ϕ a specification given as a TWTL formula. The procedure to synthesize an optimal control policy by minimizing the temporal relaxation has three steps:

1) constructing the annotated DFA $\mathcal{A}_\infty$ corresponding to ϕ,
2) constructing the synchronous product $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\infty$ between the transition system $\mathcal{T}$ and the annotated DFA $\mathcal{A}_\infty$,
3) computing the optimal policy on $\mathcal{P}$ using Alg. 14 and generating the optimal trajectory of $\mathcal{T}$ from the optimal trajectory of $\mathcal{P}$ by projection,

where the synchronous product $\mathcal{P}$ is defined as follows:

Definition VIII.2 (Product Automaton)**.** *Given a TS $\mathcal{T} = (X, x_0, \Delta, AP, h)$ and a DFA $\mathcal{A} = (S_\mathcal{A}, s_0, 2^{AP}, \delta_\mathcal{A}, F_\mathcal{A})$, their product automaton, denoted by $\mathcal{P} = \mathcal{T} \times \mathcal{A}$, is a tuple $\mathcal{P} = (S_\mathcal{P}, p_0, \Delta_\mathcal{P}, F_\mathcal{P})$ where:*

- *$p_0 = (x_0, s_0)$ is the initial state;*
- *$S_\mathcal{P} \subseteq X \times S_\mathcal{A}$ is a finite set of states that are reachable from the initial state: for every $(x^*, s^*) \in S_\mathcal{P}$, there exists a sequence of $\mathbf{x} = x_0 x_1 \dots x_n x^*$, with $x_k \rightarrow_\mathcal{T} x_{k+1}$ for all $0 \leq k < n$ and $x_n \rightarrow_\mathcal{T} x^*$, and a sequence $\mathbf{s} = s_0 s_1 \dots s_n s^*$ such that s_0 is the initial state of $\mathcal{A}$, $s_k \overset{h(x_{k+1})}{\rightarrow}_\mathcal{B} s_{k+1}$ for all $0 \leq k < n$ and $s_n \overset{h(x^*)}{\rightarrow}_\mathcal{T} s^*$;*

- $\Delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$ *is the set of transitions defined by:* $((x,s),(x',s')) \in \Delta_{\mathcal{P}}$ *iff* $x \to_{\mathcal{T}} x'$ *and* $s \overset{h(x')}{\to}_{\mathcal{B}} s'$*;*
- $F_{\mathcal{P}} = (X \times F_{\mathcal{A}}) \cap S_{\mathcal{P}}$ *is the set of accepting states of* $\mathcal{P}$*.*

A transition in $\mathcal{P}$ is also denoted by $(x,s) \to_{\mathcal{P}} (x',s')$ if $((x,s),(x',s')) \in \Delta_{\mathcal{P}}$. A trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\dots$ of $\mathcal{P}$ is an infinite sequence, where $(x_0,s_0) = p_0$ and $(x_k,s_k) \to_{\mathcal{P}} (x_{k+1},s_{k+1})$ for all $k \geq 0$. A trajectory of $\mathcal{P} = \mathcal{T} \times \mathcal{A}$ is said to be accepting if and only if it ends in a state that belongs to the set of final states $F_{\mathcal{P}}$. It follows by construction that a trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\dots$ of $\mathcal{P}$ is accepting if and only if the trajectory $s_0 s_1 \dots$ is accepting in $\mathcal{A}$. As a result, a trajectory of $\mathcal{T}$ obtained from an accepting trajectory of $\mathcal{P}$ satisfies the given specification encoded by $\mathcal{A}$. We denote the projection of a trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\dots$ onto $\mathcal{T}$ by $\gamma_{\mathcal{T}}(\mathbf{p}) = x_0 x_1 \dots$.

Before we present the details of the proposed algorithm, we want to point out that completeness may be decided easily by using the product automaton $\mathcal{P}$. That is, testing if there exists a temporal relaxation such that a satisfying policy in $\mathcal{T}$ may be synthesized can be performed very efficiently as shown by the following theorem.

Theorem VIII.1. *Let ϕ be a TWTL formula and $\mathcal{T}$ be a finite transition system. Deciding if there exists a finite $\boldsymbol{\tau} \in \mathbb{Z}^m$ and a trajectory $\mathbf{x}$ of $\mathcal{T}$ such that $\mathbf{o} \models \phi(\boldsymbol{\tau})$, can be performed in $O(|\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$, where m is the number of* within *operators in ϕ, $\mathcal{A}_\infty$ is the annotated DFA corresponding to ϕ, $\mathbf{o}$ is the output trajectory induced by $\mathbf{x}$, and Δ and $\delta_{\mathcal{A}_\infty}$ are the sets of transitions of $\mathcal{T}$ and $\mathcal{A}_\infty$, respectively.*

Remark VIII.2. *The complexity in Thm. VIII.1 is independent of the deadlines of the* within *operators ϕ.*

Proof: The result follows immediately using Dijkstra's algorithm on the product automaton $\mathcal{P}$. ∎

Note that Dijkstra's algorithm may not necessarily provide an optimal trajectory of $\mathcal{T}$ with respect to the minimum maximum temporal relaxation of the induced output word. Thus, we present a Dijkstra-based procedure to compute an optimal policy using the product automaton $\mathcal{P}$. The proposed solution is presented in Alg. 14, which describes a recursive procedure over an annotated AST tree T.

Algorithm 14: Policy synthesis – $policy(T, \mathcal{P})$

Input: T – the annotation AST tree
Input: $\mathcal{P}$ – product automaton

1 **if** ϕ_T *is primitive* **then**
2 $\quad M = \{\mathbf{p} \mid b(\mathbf{p}) \in T.I, e(\mathbf{p}) \in T.F\}$
3 $\quad \tau^*[\mathbf{p}] = -\infty, \forall \mathbf{p} \in M$
4 **else if** $T.op = [\,]^{[a,b]} \wedge \phi_{T.left}$ *is primitive* **then**
5 $\quad M = \{\mathbf{p} \mid b(\mathbf{p}) \in T.I, e(\mathbf{p}) \in T.F\}$
6 $\quad \tau^*[\mathbf{p}] = |\mathbf{p}| - b, \forall \mathbf{p} \in M$
7 **else if** $T.op = [\,]^{[a,b]} \wedge \phi_{T.left}$ *is not primitive* **then**
8 $\quad M_{ch}, \tau^{max}_{ch} = policy(T.left, \mathcal{P})$
9 $\quad M = \{p_i \overset{a}{\to} p \overset{*}{\to} p' \mid p_i \in T.I, p \overset{*}{\to} p' \in M_{ch}\}$
10 $\quad \tau^*[\mathbf{p}] = \max\{|\mathbf{p}| - b, \tau^*_{ch}[\mathbf{p}]\}, \forall \mathbf{p} \in M$
11 **else if** $T.op = \cdot$ **then**
12 $\quad M_L, \tau^*_L = policy(T.left, \mathcal{P})$
13 $\quad M_R, \tau^*_R = policy(T.right, \mathcal{P})$
14 $\quad M = \{\mathbf{p}_1 \cdot \mathbf{p}_2 \mid \mathbf{p}_1 \in M_L, \mathbf{p}_2 \in M_R, e(\mathbf{p}_1) \to_{\mathcal{P}} b(\mathbf{p}_2)\}$
15 $\quad \tau^*[\mathbf{p}] = \max\{\tau^*_L(\mathbf{p}), \tau^*_R(\mathbf{p})\}, \forall \mathbf{p} \in M$
16 **else if** $T.op = \vee$ **then**
17 $\quad M_L, \tau^*_L = policy(T.left, \mathcal{P})$
18 $\quad M_R, \tau^*_R = policy(T.right, \mathcal{P})$
19 $\quad M = M_L \cup M_R$
20 $\quad \tau^*[\mathbf{p}] = \begin{cases} \tau^*_L[\mathbf{p}] & \mathbf{p} \in M \setminus M_R \\ \tau^*_R[\mathbf{p}] & \mathbf{p} \in M \setminus M_L \\ \min\{\tau^*_L[\mathbf{p}], \tau^*_R[\mathbf{p}]\} & \mathbf{p} \in M_L \cap M_R \end{cases}$
21 **else if** $T.op = \wedge$ **then**
22 $\quad M_L, \tau^*_L = policy(T.left, \mathcal{P})$
23 $\quad M_R, \tau^*_R = policy(T.right, \mathcal{P})$
24 $\quad M = M_L \cap M_R$
25 $\quad \tau^*[\mathbf{p}] = \max\{\tau^*_L(\mathbf{p}), \tau^*_R(\mathbf{p})\}, \forall \mathbf{p} \in M$
26 **return** (M, τ^*)

The recursive procedure in Alg. 14 has six cases. The first case (lines 1-3) corresponds to a primitive formula. In this case, there are no deadlines to relax since the formula does not contain any *within* operators. Thus, solutions (if any exist) can be computed using Dijkstra's algorithm. The next two cases treat the *within* operators. In the former case (lines 4-5), the enclosed formula is a primitive formula and the only deadline which must be optimized is the one associated with the current *within* operator. In the latter case (lines 7-10), the enclosed formula is not primitive. Therefore, there are multiple deadlines that must be considered. To optimize the temporal relaxation $|\cdot|_{TR}$, we take the maximum between the previous maximum temporal relaxation and the current temporal relaxation (line 10). The fourth case (lines 11-15) handles the concatenation operator. First, the paths and the corresponding temporal relaxations are computed for the left and the right subformulae in lines 12 and 13, respectively. Afterwards, the paths satisfying the left subformula are concatenated to the paths satisfying the right formula. However, the concatenation of paths p_L and p_R is restricted to pairs which have the following property: there exists a transition in $\mathcal{P}$ between the last state of p_L and the first state in p_R. The temporal relaxation of the concatenation of two paths is the maximum between the temporal relaxations of the two paths (line 15). The next case is associated with the disjunction operator (lines 16-20). As in the concatenation case, first the paths satisfying the left M_L and the right M_R subformulae are computed in lines 17 and 18, respectively. The set corresponding to the disjunction of the two formulae is the union of the two sets because the paths must satisfy either one of the two subformulae. The temporal relaxation of a path p in the union is computed as follows (line 20): (a) if a path is only in the

left, $\mathbf{p} \in M_L \setminus M_R$, or only in the right set, $\mathbf{p} \in M_R \setminus M_L$, then the temporal relaxation is $\tau_L^*[\mathbf{p}]$ or $\tau_R^*[\mathbf{p}]$, respectively; (b) the path is in both sets, $\mathbf{p} \in M_L \cap M_R$, then the temporal relaxations is the minimum of the two previously computed ones, $\min\{\tau_L^*[\mathbf{p}], \tau_R^*[\mathbf{p}]\}$. In the case (a), $\mathbf{p}$ satisfies only one subformula and, therefore, only one temporal relaxation is available. In the case (b), $\mathbf{p}$ satisfies both subformulae. Because only one is needed, the subformula that yields the minimum temporal relaxation is chosen, i.e., the minimum between the two temporal relaxations. The last case handles the conjunction operator (lines 21-25). As in the previous two cases, the paths satisfying the left and the right subformulae are computed first (lines 22-23). Then the intersection of the two sets is computed as the set of paths satisfying the conjunctions because the paths must satisfy both subformulae. The temporal relaxations of the paths in the intersections are computed as the maxima between the previously computed temporal relaxations for the left and the right subformulae.

Note that considering primitive formulae in Alg. 14, instead of traversing the AST all the way to the leaves, optimizes the running time and the level of recursion of the algorithm.

A very important property of Alg. 14 is that its complexity does not depend on the deadlines associated with the *within* operators of the TWTL specification formula ϕ. This is an immediate consequence of the DFA construction proposed in Sec VII. Moreover, it follows from Remark IV.3 that the completeness with respect to ϕ (unrelaxed) may also be decided independently of the values of the deadline values. Formally, we have the following results.

Theorem VIII.3. *Let ϕ be a TWTL formula and $\mathcal{T}$ be a finite transition system. Synthesizing a trajectory $\mathbf{x}$ of $\mathcal{T}$ such that $\mathbf{o} \models \phi(\boldsymbol{\tau})$ and $|\boldsymbol{\tau}|_{TR}$ is minimized can be performed in $O(|\phi| \cdot |\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$, where $\boldsymbol{\tau} \in \mathbb{Z}^m$, m is the number of* within *operators in ϕ, $\mathcal{A}_\infty$ is the annotated DFA corresponding to $\phi(\infty)$, $\mathbf{o}$ is the output trajectory induced by $\mathbf{x}$, and Δ and $\delta_{\mathcal{A}_\infty}$ are the sets of transitions of $\mathcal{T}$ and $\mathcal{A}_\infty$, respectively.*

Proof: The worst-case complexity of Alg. 14 is achieved when the TWTL formula ϕ has the form of primitive formulae enclosed by *within* operators and then composed by either the conjunction, disjunction, and concatenation operators.

The recursive algorithm stops when it encounters the primitive formulae and executes Dijkstra's algorithm that takes at most $O(|\Delta_{\mathcal{P}}|) = O(|\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$ time. Since the recursion is performed with respect to an AST T of ϕ, the algorithm processes each operator only once. The complexity bound follows because the size of the set of paths M returned by the algorithm is at most the sum of the sized of the sets corresponding to the left and the right sets M_L and M_R, respectively. Thus, we obtain the bound $O(|\phi| \cdot |\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$ by summing up the time complexity over all nodes of T. ■

Corollary VIII.4. *Let ϕ be a TWTL formula and $\mathcal{T}$ be a finite transition system. Deciding if there exists a trajectory $\mathbf{x}$ of $\mathcal{T}$ such that $\mathbf{o} \models \phi$ can be performed in $O(|\phi| \cdot |\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$, where $\mathcal{A}_\infty$ is the annotated DFA corresponding to ϕ, $\mathbf{o}$ is the output trajectory induced by $\mathbf{x}$, and Δ and $\delta_{\mathcal{A}_\infty}$ are the sets of transitions of $\mathcal{T}$ and $\mathcal{A}_\infty$, respectively.*

Proof: It follows from Thm. VIII.3 and Remark IV.3. ■

C. Verification

The procedure described in Alg. 15 solves the verification problem of a transition system $\mathcal{T}$ against all relaxed versions of a TWTL specification First, the annotated DFA $\mathcal{A}_\infty$ corresponding to ϕ is computed (line 1). Then a trap state $\bowtie$ is added in line 2 (see Alg. 6 for details). The transition system $\mathcal{T}$ is composed with the DFA $\mathcal{A}_\infty$ to produce the product automaton $\mathcal{P}$ (line 3). Lastly, it is checked if a state in $\mathcal{P}$ reachable from the initial state p_0 exists such that its DFA component is the trap state $\bowtie$ (lines 4-5).

Algorithm 15: Verification

Input: $\mathcal{T}$ – transition system
Input: ϕ – TWTL specification
Output: Boolean value

1 $\mathcal{A}_\infty \leftarrow translate(\phi; \inf\ = \top)$
2 add trap state $\bowtie$ to $\mathcal{A}_\infty$
3 $\mathcal{P} \leftarrow \mathcal{T} \times \mathcal{A}_\infty$
4 **if** $\exists x \in X$ *s.t.* $p_0 \rightarrow_{\mathcal{P}} (x, \bowtie)$ **then return** $\bot$
5 **else return** $\top$

D. Learning deadlines from data

In this section, we present a simple heuristic procedure to infer deadlines from a finite set of labeled traces such that the misclassification rate is minimized. Let ϕ be a TWTL formula and $\mathcal{L}_p$ and $\mathcal{L}_n$ be two finite sets of words labeled as positive and negative examples, respectively. The misclassification rate is $|\{w \in L_p \mid w \not\models \phi(\boldsymbol{\tau})\}| + |\{w \in L_n \mid w \models \phi(\boldsymbol{\tau})\}|$, where $\phi(\boldsymbol{\tau})$ is a feasible $\boldsymbol{\tau}$-relaxation of ϕ. The terms of the misclassification rate are the false negative and false positive rates, respectively.

The procedure presented in Alg. 16 uses Alg. 11 to compute the tightest deadlines for each trace. Then each deadline is determined in a greedy way such that the misclassification rate is minimized. The heuristic in Alg. 11 is due to the fact that each deadline is considered separately from the others. However, the deadlines are not independent with respect to the minimization of the misclassification rate.

Notice that the algorithm constructs $\mathcal{A}_\infty$ only once at line 1. Then the automaton is used in the $tr(\cdot)$ function to compute the temporal relaxation of each trace, lines 2-3. Thus, the procedure avoids building $\mathcal{A}_\infty$ for each trace.

In Alg. 16, m denotes the number of *within* operators and $\mathbf{b}$ is the m-dimensional vector of deadlines associated with each *within* operator in the TWTL formula ϕ. We assume that the order of the *within* operators is given by the post-order traversal of $AST(\phi)$, i.e., recursively traversing the children nodes first and then the node itself.

The complexity of the learning procedure is $O\big(2^{|\phi|+|AP|} + (|\mathcal{L}_p| + |\mathcal{L}_n|) \cdot l_m \cdot |\phi| + m \cdot (|\mathcal{L}_p| + |\mathcal{L}_n|)\big)$, where: (a) the first

Algorithm 16: Parameter learning

Input: $\mathcal{L}_p$ – set of positive traces
Input: $\mathcal{L}_n$ – set of negative traces
Input: ϕ – template TWTL formula
Output: d – the vector of deadlines

1 $\mathcal{A}_\infty \leftarrow translate(\phi; \inf\ = \top)$
2 $D_p \leftarrow \{tr(p, \mathcal{A}_\infty) + \mathbf{b} \mid p \in \mathcal{L}_p\}$
3 $D_n \leftarrow \{tr(p, \mathcal{A}_\infty) + \mathbf{b} \mid p \in \mathcal{L}_n\}$
4 $\mathbf{d} \leftarrow (-\infty, -\infty, \ldots, -\infty)$ `// m-dimensional`
5 **for** $k \in \{1, \ldots, m\}$ **do**
6 $\quad D_k \leftarrow \{d[k] \mid d \in D_p\}$
7 $\quad \mathbf{d}[k] \leftarrow \arg\min_{d \in D_k} \left(\left| D^k_{FP}(d) \right| + \left| D^k_{FN}(d) \right| \right)$, where
8 $\quad\quad D^k_{FP}(d) \leftarrow \{\mathbf{d}'[k] \mid \mathbf{d}'[k] > d, \mathbf{d}' \in D_n\}$
9 $\quad\quad D^k_{FN}(d) \leftarrow \{\mathbf{d}'[k] \mid \mathbf{d}'[k] \leq d, \mathbf{d}' \in D_p\}$
10 **return** d

term is the complexity of constructing $\mathcal{A}_\infty$ (line 1); (b) the second term corresponds to computing the tight deadlines for all traces positive and negative in lines 2 and 3, respectively; (c) the third term is the complexity of the for loop, which computes each deadline separately in a greedy fashion (lines 5-9). The maximum length of a trace (positive or negative) is denoted by l_m in the complexity formula.

IX. TWTL Python Package

We provide a Python 2.7 implementation named PyTWTL of the proposed algorithms based on LOMAP [35], ANTLRv3 [28] and networkx [13] libraries. PyTWTL implementation is released under the GPLv3 license and can be downloaded from *hyness.bu.edu/twtl*. The library may be used to:

1) construct a DFA $\mathcal{A}_\phi$ and a annotated DFA $\mathcal{A}_\infty$ from a TWTL formula ϕ;
2) monitor the satisfaction of a TWTL formula ϕ;
3) monitor the satisfaction of an arbitrary relaxation of ϕ, i.e., $\phi(\infty)$;
4) compute the temporal relaxation of a trace with respect to a TWTL formula;
5) compute a satisfying control policy with respect to a TWTL formula ϕ;
6) compute a minimally relaxed control policy with respect to a TWTL formula ϕ, i.e., for $\phi(\boldsymbol{\tau})$ such that $|\boldsymbol{\tau}|_{TR}$ is minimal.

The parsing of TWTL formulae is performed using ANTLRv3 framework. We provide grammar files which may be used to port to generate lexers and parsers for other programming languages such as Java, C/C++, Ruby. To support Python 2.7, we used version 3.1.3 of ANTLRv3 and the corresponding Python runtime ANTLR library, which we included in our distribution for convenience.

X. Case Studies

In this section, we present some examples highlighting the solutions for the verification, synthesis and learning problems. First, we show the automaton construction procedure on a TWTL formula and how the tight deadlines are inferred for a given trace. Then, we consider an example involving a robot whose motion is modeled as a TS. The policy computation algorithm is used to solve a path planning problem with rich specifications given as TWTL formulae. The procedure for performing verification, i.e., all robot trajectories satisfy a given TWTL specification, is also shown. Finally, the performance of the heuristic learning algorithm is demonstrated on a simple example.

A. Automata Construction and Temporal Relaxation

Consider the following TWTL specification over the set of atomic propositions $AP = \{A, B, C, D\}$:

$$\phi = [H^2 A]^{[0,6]} \cdot ([H^1 B]^{[0,3]} \vee [H^1 C]^{[1,4]}) \cdot [H^1 D]^{[0,6]} \quad (19)$$

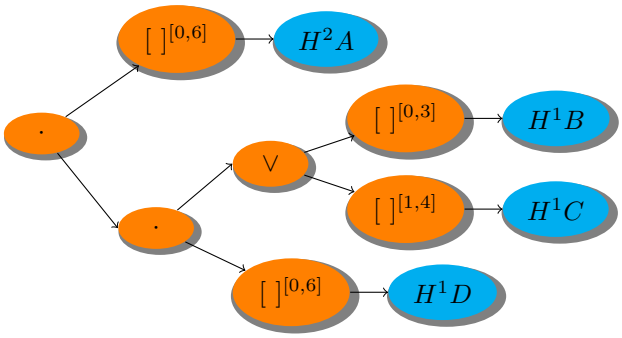
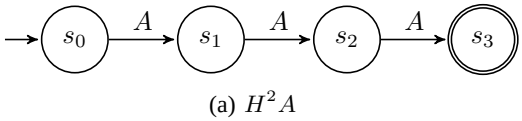


Fig. 2: The AST corresponding to the TWTL formula in Eq. (19).

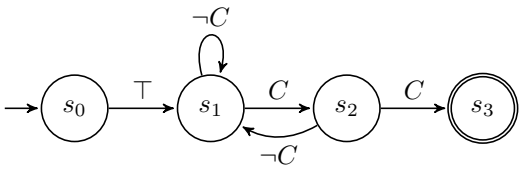
An AST of formula ϕ is shown in Fig. 2. The TWTL formula ϕ is converted to an annotated DFA $\mathcal{A}_\infty$ using Alg. 1. The procedure recursively constructs the DFA from the leafs of the AST to the root. A few processing steps are shown in Fig. 3. The construction of DFA corresponding to a leaf, i.e., a *hold* operator, is straightforward, see Fig. 3a. Next, the transformation corresponding to a *within* operator is shown in Fig. 3b. Note that the delay of one time unit is due to the lower bound of the time window of the *within* operator. Also, note that the automaton restarts on symbols that block the DFA corresponding to the inner formula $H^1 C$.

The next two figures, Fig. 3c and Fig. 3d, show the translation of the disjunction operator. Specifically, Fig. 3c, shows the product DFA corresponding to the disjunction without merging the final states. Since none of the final states have outgoing transitions, see Corr. VI.9, and they can be merged into a single final state, see Fig. 3d. However, we still need to keep track of which subformula of the disjunctions holds. The annotation variable $T.choices$, introduced in Sec. VII-C2, stores this information as

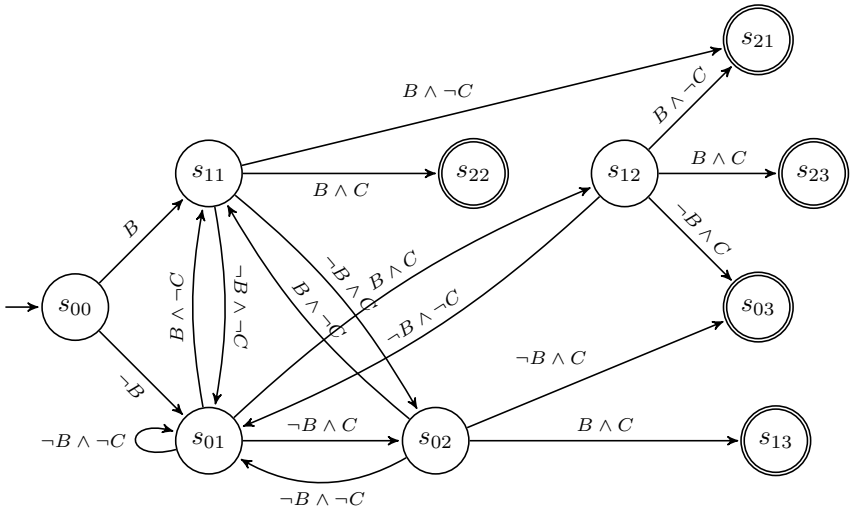
$$\begin{cases} L = \{(s_{11}, B \wedge \neg C), (s_{11}, B \wedge C), (s_{12}, B \wedge \neg C)\}, \\ R = \{(s_{02}, \neg B \wedge C), (s_{02}, B \wedge C), (s_{12}, \neg B \wedge C)\}, \\ B = \{(s_{12}, B \wedge C)\}. \end{cases} \quad (20)$$



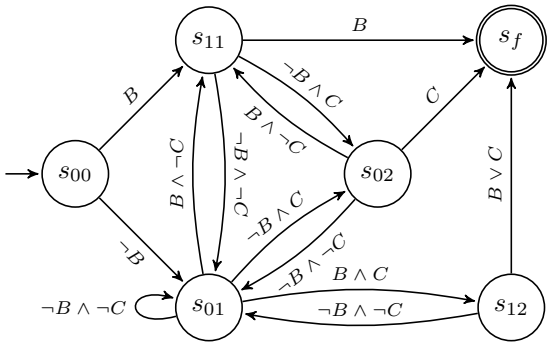
(a) $H^2 A$



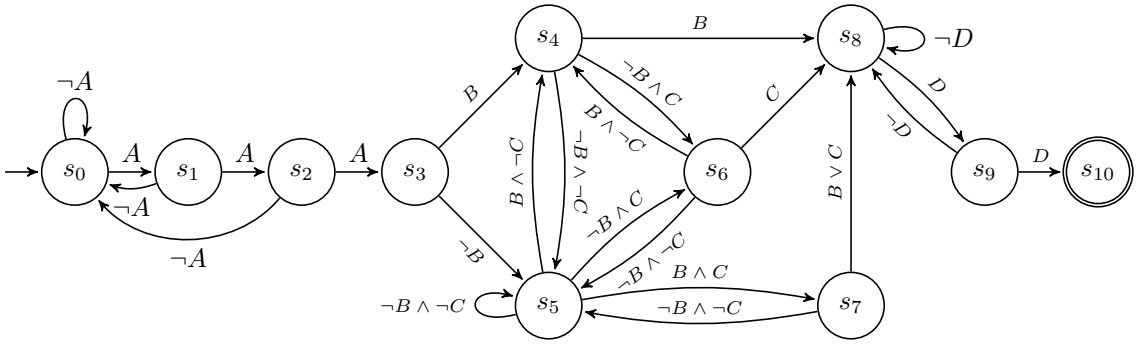
(b) $[H^1 C]^{[1,\ ]}$



(c) $[H^1 B]^{[0,\ ]} \vee [H^1 C]^{[1,\ ]}$



(d) $[H^1 B]^{[0,\ ]} \vee [H^1 C]^{[1,\ ]}$



(e) $[H^2 A]^{[0,\ ]} \cdot ([H^1 B]^{[0,\ ]} \vee [H^1 C]^{[1,\ ]}) \cdot [H^1 D]^{[0,\ ]}$

Fig. 3: Annotated automata corresponding to subformulae of the TWTL specification in Eq. (19).

Notice that the tuples in Eq. (20) correspond to the ingoing edges of the final states in the DFA from Fig. 3c. Finally, the DFA corresponding to the overall specification formula ϕ is shown in Fig. 3e.

Let $\phi_A = [H^2 A]^{[0,6]}$, $\phi_B = [H^1 B]^{[0,3]}$, $\phi_C = [H^1 C]^{[1,4]}$, and $\phi_D = [H^1 D]^{[0,6]}$ be subformulae of ϕ associated with the *within* operators. The annotation data for these subformulae is shown in the following table.

Subformula	$T.I$	$T.F$
ϕ	$\{s_0\}$	$\{s_{10}\}$
ϕ_A	$\{s_0\}$	$\{s_3\}$
ϕ_B	$\{s_3, s_5, s_6\}$	$\{s_8\}$
ϕ_C	$\{s_3\}$	$\{s_3\}$
ϕ_D	$\{s_8\}$	$\{s_{10}\}$

Consider the following word over the alphabet $\Sigma = 2^{AP}$:

$$\boldsymbol{\sigma} = \epsilon, \{A\}, \{A\}, \{A\}, \epsilon, \{B, C\}, \{B, C\}, \epsilon, \{D\}, \{D\} \quad (21)$$

where ϵ is the empty symbol. The following table shows the stages of Alg. 11 as the symbols of the word $\boldsymbol{\sigma}$ are processed:

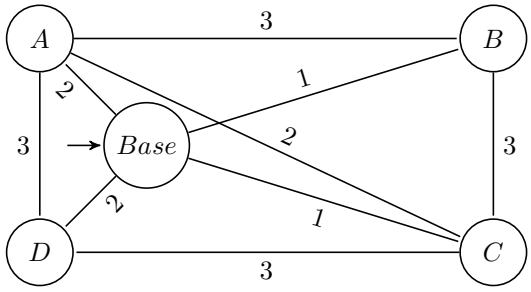
No.	Symbol	State	ϕ_A	ϕ_B	ϕ_C	ϕ_D
Init		s_0	$(\top, \bot, 0)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
0	ϵ	s_0	$(\top, \bot, 1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
1	$\{A\}$	s_1	$(\top, \bot, 2)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
2	$\{A\}$	s_2	$(\top, \bot, 3)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
3	$\{A\}$	s_3	$(\bot, \top, 3)$	$(\top, \bot, 0)$	$(\top, \bot, 0)$	$(\bot, \bot, -1)$
4	ϵ	s_5	$(\bot, \top, 3)$	$(\top, \bot, 1)$	$(\top, \bot, 1)$	$(\bot, \bot, -1)$
5	$\{B, C\}$	s_7	$(\bot, \top, 3)$	$(\top, \bot, 2)$	$(\top, \bot, 2)$	$(\bot, \bot, -1)$
6	$\{B, C\}$	s_8	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\top, \bot, 0)$
7	ϵ	s_8	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\top, \bot, 1)$
8	$\{D\}$	s_9	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\top, \bot, 2)$
9	$\{D\}$	s_{10}	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$

where each 3-tuple in last four columns represents the annotation variables $T.ongoing$, $T.done$ and $T.steps$, respectively. The temporal relaxation for $\boldsymbol{\sigma}$ can be extracted from the values in the last row by subtracting the deadlines of the *within* operators from them. Thus, the vector of tightest τ values is $(-3, -1, -2, -3)$. However, because ϕ_B and ϕ_C are in disjunction, we have the temporal relaxation $\boldsymbol{\tau} = (-3, -\infty, -2, -3)$, where we choose to ignore the subformula containing ϕ_B. Thus, the maximum temporal relaxation is $|\boldsymbol{\tau}|_{TR} = -2$.

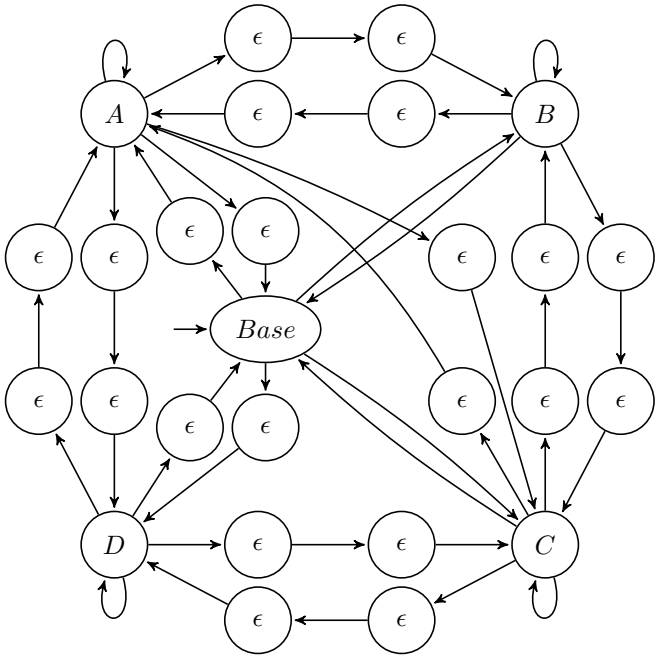
B. Control Policy Synthesis

Consider a robot moving in an environment represented as the finite graph shown in Fig. 4a. The nodes of the graph represent the points of interest, while the edges indicate the possibility of moving the robot between the edges' endpoints. The numbers associated with the edges represent the travel times, and we assume that all the travel times are integer multiples of a time step Δt. The robot may also stay at any of the points of interest.

The motion of the robot is abstracted as a transition system $\mathcal{T}$, which is obtained from the finite graph by splitting each edge into a number of transitions equal to the corresponding edge's travel time. The generated transition system thus has 27 states and 67 transitions and is shown in Fig. 4b.



(a) An environment with five points of interest, $Base$ A, B, C, and D. The edges indicate the existence of paths between their endpoints, while the associated numbers represent the travel times of the edges. The robot may stay at a region of interest.



(b) The transition system $\mathcal{T}$ obtained from the environment graph shown in Fig. 4a.

Fig. 4: The environment where the robot operates and its abstraction $\mathcal{T}$.

Consider the TWTL specification ϕ from Eq. (19). The product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\infty$ is constructed, where $\mathcal{A}_\infty$ is the annotated DFA corresponding to $\phi(\infty)$ shown in Fig. 3e. The product automaton $\mathcal{P}$ has 204 states and 378 transitions. The control policy computed by using Alg. 14 is

$$\mathbf{x} = Base, A, A, A, C, C, Base, D, D, \quad (22)$$

which generates the output word

$$\boldsymbol{\sigma} = \epsilon, \epsilon, \{A\}, \{A\}, \{A\}, \epsilon, \{C\}, \{C\}, \epsilon, \epsilon, \{D\}, \{D\}. \quad (23)$$

The minimum temporal relaxation for $\boldsymbol{\sigma}$ is $|\boldsymbol{\tau}|_{TR} = -2$, where $\boldsymbol{\tau} = (-2, -\infty, -2, -3)$ is the minimal temporal relaxation vector associated with $\boldsymbol{\sigma}$.

C. Verification

In the verification problem, we are concerned with checking for the existence of relaxed specifications for every possible run of a transition system.

To illustrate this problem, consider the transition system in Fig. 5 and the following two TWTL specifications:

$$\phi_1 = [H^1 A]^{[1,2]} \tag{24}$$

$$\phi_2 = [H^1 \neg B]^{[1,2]} \tag{25}$$

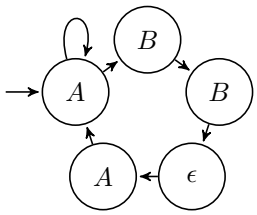


Fig. 5: A simple transition system $\mathcal{T}^{simple}$.

To check the transition system $\mathcal{T}^{simple}$ against the two specifcations, we can use Alg. 15. It is straightforward that the procedure will return true for ϕ_1, because every run of $\mathcal{T}^{simple}$ satisfies $\phi_1(3) = [H^1]^{[1,2+3]}$. Note that the runs of the transition system may not need to satisfy the original specification as the satisfaction of a relaxed version is sufficient. Similarly, Alg. 15 returns false for ϕ_2, because there exists a run of $\mathcal{T}^{simple}$ that does not satisfy the specification, e.g., $\mathbf{x} = A, B, B$.

An important conclusion highlighted by the two examples is that the verification problem proposed in this paper is concerned with checking a system against the logical structure of a specification and not against any particular time bounds. This might be useful in situation where the deadlines of the specification are not known *a priori*, but the logical structure of the specification is.

D. Learning deadlines from data

In the previous two cases, we use the TWTL specifications in conjunction with problems involving infinite sets of words encoded as transition systems. However, it is often the case that only finite sets of (output) trajectories are available. In this section, we give a simple example of the learning problem presented in Sec. V.

Consider the specification $\phi_{learn} = [H^1]^{[0,d_1]} \cdot [H^2 B]^{[0,d_2]}$ with unknown deadlines and the following set of labeled trajectories, where C_p and C_n are the positive and negative example labels, respectively:

Word	Label	Deadlines
$\boldsymbol{\sigma}_1$ =$\{A\}$,$\{A\}$,$\{A\}$,$\{B\}$,$\{B\}$,$\{B\}$,$\{B\}$,ϵ	C_p	$(2,3)$
$\boldsymbol{\sigma}_2$ =ϵ, $\{A\}$,$\{A\}$,ϵ, $\{B\}$,$\{B\}$,$\{B\}$,ϵ	C_p	$(2,3)$
$\boldsymbol{\sigma}_3$ =$\{B\}$,ϵ, $\{A\}$,$\{A\}$,$\{B\}$,$\{B\}$,$\{B\}$,$\{B\}$	C_n	$(3,2)$
$\boldsymbol{\sigma}_4$ =ϵ, $\{A\}$,$\{A\}$,ϵ, ϵ, $\{B\}$,$\{B\}$,$\{B\}$	C_n	$(2,4)$

The last column in the above table shows the tight deadlines obtained in lines 2 and 3 of Alg. 16. Next, the learning algorithm computes the heuristic sets D^k_{FP} and D^k_{FN}, $k \in \{d_1, d_2\}$, of false positive and false negative trajectories, respectively:

Deadline	Value	D^k_{FP}	D^k_{FN}	$\lvert D^k_{FP}\rvert + \lvert D^k_{FN}\rvert$
d_1	2	$\{\boldsymbol{\sigma}_4\}$	$\emptyset$	1
d_1	3	$\{\boldsymbol{\sigma}_3, \boldsymbol{\sigma}_4\}$	$\emptyset$	2
d_2	2	$\{\boldsymbol{\sigma}_3\}$	$\{\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2\}$	3
d_2	3	$\{\boldsymbol{\sigma}_3\}$	$\emptyset$	1
d_2	4	$\{\boldsymbol{\sigma}_3, \boldsymbol{\sigma}_4\}$	$\emptyset$	2

Finally, Alg. 16 chooses the deadline pair $\mathbf{d} = (d_1, d_2) = (2, 3)$ that has the lowest heuristic misclassification rate, $\lvert D^k_{FP}\rvert + \lvert D^k_{FN}\rvert$ shown in the last column of the above table, for d_1 and d_2, respectively. An important observation is that the inferred formula $\phi^{\mathbf{d}}_{learn} = [H^1 A]^{[0,2]} \cdot [H^2 B]^{[0,3]}$ has zero as actual misclassification rate. The discrepancy between the values in the table and the actual value of the final misclassification rate are due to the heuristic of synthesizing each deadline separately. Thus, the heuristic procedure in Alg. 16 ignores the temporal and logical structure of the template TWTL formula which may lead to suboptimal performance, i.e., misclassification rate.

XI. Conclusion

In this paper, we introduced the time window temporal logic (TWTL), which is a linear-time logic encoding sets of discrete-time bounded specifications. Different from other temporal logics, TWTL has an explicit concatenation operator, which enables the compact representation of serial tasks mostly prevalent in robotics and control applications. Such a compact representation significantly reduces the complexity of constructing the automaton for the accepting language. In this paper, we also discussed the temporal relaxation of TWTL formulae and provided provably-correct algorithms to construct a compact automaton representing all temporally relaxed formulae of a given TWTL formula. Stemming from the definition of temporal relaxation, we formulated some problems related to verification, synthesis, and learning. We demonstrated the potential of TWTL and its relaxation on these problems. Finally, we also developed a Python package to solve the verification, synthesis, and learning problems.

XII. Appendices

A. Proof of Prop. VI.3

Proof: Let $\big(L_1, \mathcal{L}_1, \mathcal{L}_1 \cdot (P(\mathcal{L}_2) \setminus \{\epsilon\})\big)$ be a partition of $P(\mathcal{L}_1 \cdot \mathcal{L}_2)$ and L be a proper subset of $\mathcal{L}_1$. Assume that there exists $w \in L$ and $w' \in \mathcal{L}_1 \setminus L$ such that $w = w'_{0,i}$, for some $i \in \{0, \ldots, |w'| - 1\}$. It follows that $w \in L_1$, because $w \neq w'$. However, this contradicts the fact that L_1 and $\mathcal{L}_1$ are disjoint.

Conversely, let $\mathcal{L}_1$ be unambiguous and consider a word $w \in P(\mathcal{L}_1 \cdot \mathcal{L}_2)$. Assume that $w \in L_1 \cap \mathcal{L}_1$. It follows that $\{w\}$ is a prefix language for $\mathcal{L}_1 \setminus \{w\}$, which contradicts with the hypothesis that $\mathcal{L}_1$ is unambiguous. Similarly, if we assume that there exists $w \in P(\mathcal{L}_1) \cap \big(\mathcal{L}_1 \cdot (P(\mathcal{L}_2) \setminus \{\epsilon\})\big)$, then there

exists $w', w'' \in \mathcal{L}_1$ such that w' is a prefix of w, w is a prefix of w'', and $|w'| < |w| \leq |w''|$. Thus, we arrive again at a contradiction with the unambiguity of $\mathcal{L}_1$. Thus, the three sets form a partition of $P(\mathcal{L}_1 \cdot \mathcal{L}_2)$. ■

B. Proof of Prop. VI.4

Proof: The proof follows by structural induction over $AST(\phi)$. The base case is trivial, since the leafs correspond to the *hold* operators. For the induction step, the result follows trivially if the intermediate node is associated with a Boolean or concatenation operator. The case of a *within* operator follows from Eq. (17) and (18) in Prop. VI.1, i.e. $[\phi(\boldsymbol{\tau})]^{[a,b+\tau_1]} \Rightarrow [\phi(\boldsymbol{\tau}')]^{[a,b+\tau_1]} \Rightarrow [\phi(\boldsymbol{\tau}')]^{[a,b+\tau_1']}$, where $a < b \in \mathbb{Z}_{\geq 0}$ and $\boldsymbol{\tau} \leq \boldsymbol{\tau}' \in \mathbb{Z}^m$. We assumed without loss of generality that the first component of the temporal relaxation vectors is assigned to the root node. ■

REFERENCES

[1] Derya Aksaray, Kevin Leahy, and Calin Belta. Distributed multi-agent persistent surveillance under temporal logic constraints. *IFAC-PapersOnLine*, 48(22):174–179, 2015.

[2] Derya Aksaray, Cristian-Ioan Vasile, and Calin Belta. Dynamic routing of energy-aware vehicles with temporal logic constraints. In *submitted to IEEE Int. Conference on Robotics and Automation (ICRA)*, 2016.

[3] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[5] C. Belta, V. Isler, and G. J. Pappas. Discrete abstractions for robot planning and control in polygonal environments. *IEEE Trans. on Robotics*, 21(5):864–874, 2005.

[6] C. Câmpeanu, II Culik, K., Kai Salomaa, and Sheng Yu. State Complexity of Basic Operations on Finite Languages. In Oliver Boldt and Helmut Jrgensen, editors, *Automata Implementation*, volume 2214 of *Lecture Notes in Computer Science*, pages 60–70. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42812-1. doi: 10.1007/3-540-45526-4_6. URL http://dx.doi.org/10.1007/3-540-45526-4_6.

[7] Igor Cizelj and Calin Belta. Control of Noisy Differential-Drive Vehicles from Time-Bounded Temporal Logic Specifications. In *IEEE Int. Conference on Robotics and Automation (ICRA)*, 2013.

[8] Jan Daciuk. Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. In *Implementation and Application of Automata*, pages 255–261. Springer, 2003.

[9] Alexandre Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, volume 8172 of *Lecture Notes in Computer Science*, pages 442–445, Hanoi, Vietnam, Oct 2013. Springer. doi: 10.1007/978-3-319-02444-8_31.

[10] Georgios E Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352, 2009.

[11] Yuan Gao, Kai Salomaa, and Sheng Yu. Transition Complexity of Incomplete DFAs. *Fundam. Inf.*, 110 (1-4):143–158, Jan 2011. ISSN 0169-2968. URL http://dl.acm.org/citation.cfm?id=2362097.2362107.

[12] Meng Guo and Dimos V. Dimarogonas. Multi-agent plan reconfiguration under local ltl specifications. *The International Journal of Robotics Research*, 34(2):218–235, 2015. doi: 10.1177/0278364914546174. URL http://ijr.sagepub.com/content/34/2/218.abstract.

[13] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. 2008.

[14] Yo-Sub Han and Kai Salomaa. State Complexity of Union and Intersection of Finite Languages. In Tero Harju, Juhani Karhumki, and Arto Lepist, editors, *Developments in Language Theory*, volume 4588 of *Lecture Notes in Computer Science*, pages 217–228. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73207-5. doi: 10.1007/978-3-540-73208-2_22. URL http://dx.doi.org/10.1007/978-3-540-73208-2_22.

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[16] Sumit K. Jha, Edmund M. Clarke, Christopher J. Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In *Proc. of the 7th Int. Conference on Computational Methods in Systems Biology*, CMSB '09, pages 218–234, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03844-0.

[17] S. Karaman and E. Frazzoli. Vehicle routing problem with metric temporal logic specifications. In *IEEE Conference on Decision and Control*, pages 3953 – 3958, December 2008.

[18] M. Kloetzer and C. Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. on Automatic Control*, 53(1): 287–297, 2008.

[19] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[20] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Trans. on Robotics*, 25(6):1370–1381, 2009.

[21] Orna Kupferman and Moshe Y Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[22] T. Latvala. Effcient model checking of safety properties. In *10th International SPIN Workshop*, Model Checking Software, pages 74–88. Springer, 2003.

[23] K. Leahy, D. Zhou, C.I. Vasile, K. Oikonomopoulos, M. Schwager, and C. Belta. Provably Correct Persistent Surveillance for Unmanned Aerial Vehicles Subject to Charging Constraints. In *Int. Symposium on Experimental Robotics (ISER)*, 2014.

[24] Scott C. Livingston, Pavithra Prabhakar, Alex B. Jose, and Richard M. Murray. Patching task-level robot controllers based on a local μ-calculus formula. In *International Conference on Robotics and Automation (ICRA)*, 2013.

[25] Eva Maia, Nelma Moreira, and Rogrio Reis. Incomplete Transition Complexity of Some Basic Operations. In Peter van Emde Boas, FransC.A. Groen, GiuseppeF. Italiano, Jerzy Nawrocki, and Harald Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science*, volume 7741 of *Lecture Notes in Computer Science*, pages 319–331. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35842-5. doi: 10.1007/978-3-642-35843-2_28. URL http://dx.doi.org/10.1007/978-3-642-35843-2_28.

[26] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[27] Zohar Manna and Amir Pnueli. Verification of concurrent programs. part i. the temporal framework. Technical report, DTIC Document, 1981.

[28] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages* . Pragmatic Bookshelf, 2007. ISBN 978-0978739256.

[29] Marco Pavone, Nabhendra Bisnik, Emilio Frazzoli, and Volkan Isler. A stochastic and dynamic vehicle routing problem with time windows and customer impatience. *Mobile Networks and Applications*, 14(3):350–364, 2009.

[30] L.I. Reyes Castro, P. Chaudhari, J. Tumova, S. Karaman, E. Frazzoli, and D. Rus. Incremental sampling-based algorithm for minimum-violation motion planning. In *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pages 3217–3224, Dec 2013. doi: 10.1109/CDC.2013.6760374.

[31] Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[32] I. Tkachev and A. Abate. Formula-free Finite Abstractions for Linear Temporal Verification of Stochastic Hybrid Systems. In *Proc. of the 16th Int. Conference on Hybrid Systems: Computation and Control*, Philadelphia, PA, April 2013.

[33] J. Tumova, A. Marzinotto, D.V. Dimarogonas, and D. Kragic. Maximally satisfying ltl action planning. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1503–1510, Sept 2014. doi: 10.1109/IROS.2014.6942755.

[34] Jana Tumova, Gavin C. Hall, Sertac Karaman, Emilio Frazzoli, and Daniela Rus. Least-violating control strategy synthesis with safety rules. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 1–10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1567-8. doi: 10.1145/2461328.2461330. URL http://doi.acm.org/10.1145/2461328.2461330.

[35] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus. Optimality and Robustness in Multi-Robot Path Planning with Temporal Logic Constraints. *Int. Journal of Robotics Research*, 32(8):889–911, 2013.

[36] C.I. Vasile and C. Belta. An Automata-Theoretic Approach to the Vehicle Routing Problem. In *Proc. of the Robotics: Science and Systems Conference (RSS)*, Berkeley, California, USA, July 2014.

[37] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding Horizon Temporal Logic Planning for Dynamical Systems. In *IEEE Conference on Decision and Control (CDC)*, pages 5997 –6004, 2009.

[38] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. Receding horizon control for temporal logic specifications. In *Proc. of the 13th Int. Conference on Hybrid Systems: Computation and Control*, pages 101–110. ACM, 2010.