Timed Trace Expressions

Luca Ciccone[1], Angelo Ferrando[2], Davide Ancona[1], and Viviana Mascardi[1]

[1] University of Genova, Genova, Italy,
4077756@studenti.unige.it,davide.ancona@unige.it,viviana.mascardi@unige.it
[2] Liverpool University, Liverpool, United Kingdom,
angelo.ferrando@liverpool.ac.uk*

Abstract Trace expressions are a compact and expressive formalism initially devised for runtime verification of multiagent systems, and then adopted for runtime verification of object oriented systems and of Internet of Things applications. In this paper we survey different logics to cope with time intervals, and we exploit the ideas underlying these logics to extend the trace expressions formalism with the explicit management of time.

Keywords: Timed Temporal Logic, Timed Trace Expressions, Runtime Verification

1 Introduction and Motivation

Alice and Bob will attend CILC 2019: they agree to meet in front of the CILC venue between 9.00 AM and 9.20 AM, and enter the building together. Bob is always on time. He knows that Alice tends to be late, so he points out that if she will not be there between 9.00 AM and 9.20 AM, he will enter, without waiting for her. Entering the venue makes sense only before 11.00. So if Alice is late, but not too late, she enters as soon as she reaches the venue. If she is too late, she gives up attending that session.

We may formally specify the agreement between Alice and Bob in the following way:

$Agreement = (\mathrm{bob\_on\_time} : \epsilon) \mid$
$(Alice\_on\_time \vee Alice\_standard\_delay \vee Alice\_except\_delay)$

where

$Alice\_on\_time = (\mathrm{alice\_on\_time} : \mathrm{alice\_and\_bob\_enter\_together} : \epsilon)$

$Alice\_standard\_delay = (\mathrm{alice\_late} : ((\mathrm{bob\_enters} : \epsilon) \mid (\mathrm{alice\_enters} : \epsilon)))$

$Alice\_except\_delay = (\mathrm{alice\_too\_late} : ((\mathrm{bob\_enters} : \epsilon) \mid (\mathrm{alice\_gives\_up} : \epsilon)))$

* Work supported by EPSRC as part of the ORCA [EP/R026173] and RAIN [EP/R026084] Robotics and AI Hubs.

We use italic to denote expressions and sub-expressions that represent the agreement, and that are defined by equations that could be recursive[3]; we use lowercase strings to denote the types of events that are expected to take place.

To grasp the intuition behind the above formalization, we need to gently introduce the operators appearing therein, and to better clarify the notion of "types of events that are expected to take place".

ev_type:*Expr* means that an event whose type is ev_type takes place before the events modelled by *Expr*. The empty expression is represented by ϵ, hence ev_type:ϵ means that after an event with type ev_type took place, nothing more should happen. The operator shuffle | applies to two expressions, and means that the events taking place in the left expression can be interleaved in whatever way with those in the right expression (but the event order within the two expressions must be preserved). The operator $\vee$ denotes mutually exclusive choice between two expressions, and $Expr_1 \cdot Expr_2$ means that after the events modelled by $Expr_1$ took place, then those in $Expr_2$ will start to take place (concatenation).

Now let us move to better shaping the meaning of bob_on_time. Bob is on time if he reaches the CILC venue between 9.00 AM and 9.20 AM; many real events match this "type", where the interval of validity must be explicitly specified. We can state that bob_on_time is characterized by the actual events to be observed and the time interval when they should be observed:
bob_on_time = ⟨ {"bob in front of CILC venue"}, [9.00 AM, 9.20 AM] ⟩.

In the same way we can define alice_on_time = ⟨ {"alice in front of CILC venue"}, [9.00 AM, 9.20 AM] ⟩, alice_late = ⟨ {"alice in front of CILC venue"}, (9.20 AM, 11.00 AM] ⟩, and alice_too_late = ⟨ {"alice in front of CILC venue"}, (11.00 AM, 12.00 PM] ⟩.

We might want to model the event of being in the right place in a more detailed way. So, for example, the events B1 = "bob in front of the main door of the CILC venue", B2 = "bob on the external stairs of the CILC venue", B3 = "bob in the main entrance of the CILC venue", might all be considered valid to state that Bob reached the CILC venue. In this case, we might define bob_on_time = ⟨ {B1, B2, B3}, [9.00 AM, 9.20 AM] ⟩.

The event of entering together should be associated with the interval [9.00 AM, 9.20 AM], while entering alone can take place the interval (9.20 AM, 11.00 AM] for both Alice and Bob; giving up takes place when Alice realizes she is too late; it holds in (11.00 AM, 12.00 PM].

Let us suppose the following events are observed, each associated with the time it was observed: ⟨ "bob is in front of the CILC venue", 9.00 AM ⟩, ⟨ "alice is in front of the CILC venue", 9.09 AM ⟩, ⟨ "alice and bob enter together", 9.12 AM ⟩. If a runtime monitor were in charge of verifying the *Agreement*, it should output "yes" after observing the events above.

The sequence ⟨ "bob is in front of the CILC venue", 9.00 AM ⟩, ⟨ "bob enters the CILC venue alone", 9.05 AM ⟩, ⟨ "alice is in front of the CILC venue", 9.15

[3] They are not in these examples, but "*Cheers* = alice_says_hello : *Cheers*" would be a perfectly legal expression.

AM $\rangle$ does not meet the *Agreement*, as Bob should not enter alone before 9.20 AM. The monitor should output "no", or raise some alarm, in this case.

Finally, we observe that the *Agreement* does not even consider the possibility for Bob to be late. If Bob does not reach the venue in time, the monitor should output "no" as well.

In this paper we present Timed Trace Expressions, an extension of Trace Expressions [3,4,5] with time constraints. Timed Trace Expressions can be used to formalize the above *Agreement* between Bob and Alice. Given that the theoretical underpinning of this extension is given by Interval Temporal Logic and Metric Interval Temporal Logic, we present a survey of Temporal Logics formalisms in Section 2. Section 3 introduces Timed Trace Expressions, and Section 4 concludes and outlines the future directions of our work.

2 Background on (Interval) Temporal Logics

Many surveys on Temporal Logics exist, starting from [10,16] and moving to more recent works like [8,14,18], which take time and intervals into account.

In this section we summarize the works more relevant to ours; the survey is driven by our goal, namely runtime verification [12] of distributed systems in general, and of multiagent systems [11,19,20] in particular.

2.1 Linear Temporal Logic, LTL

Linear Temporal Logic (LTL, [15]) does not deal with discrete time intervals, but introducing its syntax and semantics is a step forward introducing Timed Temporal Logic and Metric Temporal Logic in the sequel. LTL finds it main application in model checking [6]. In order to use LTL for runtime verification, it was extended to LTL$_3$ [7].

LTL Syntax. Let AP be a set of atomic propositions:

$$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\, \phi \mid \bigcirc\, \phi \mid \phi_1 \bigcup \phi_2$$

Where:

- $a \in AP$
- $\bigcirc$ is for *next*
- $\bigcup$ is for *until*

LTL formulae are evaluated on a sequence of states. Additional operators can be obtained by combining the ones above, like the *necessity* and *possibility* modal operators that can be expressed from a temporal point of view as:

- $\mathbf{F}\phi\ (\Diamond\ \phi) \equiv true \bigcup \phi$
- $\mathbf{G}\phi\ (\Box\ \phi) \equiv \neg(true \bigcup \neg\ \phi)$

LTL Semantics. Let Σ be an alphabet such that $\Sigma = 2^{AP}$. We consider an infinite trace $w = a_0 a_1 a_2 ... \in \Sigma^\omega$. Let ϕ_i be an LTL formula. By w[j...] we identify the suffix of w starting in position j, namely $a_j a_{j+1}...$

- $w \models true$
- $w \models a$ iff $a \in a_0$
- $w \models \phi_1 \wedge \phi_2$ iff $w \models \phi_1$ and $w \models \phi_2$
- $w \models \neg\, \phi$ iff $w \not\models \phi$
- $w \models \bigcirc\, \phi$ iff $w[1...] \models \phi$
- $w \models \phi_1 \bigcup \phi_2$ iff $\exists j \geq 0$ such that $w[j...] \models \phi_2$ and $w[i...] \models \phi_1$ for all 0 $\leq i < j$

Paths can also satisfy the necessity and possibility operators.

- $w \models \mathbf{F}\phi$ iff $\exists j \geq 0$ such that $w[j...] \models \phi$
- $w \models \mathbf{G}\phi$ iff $\forall j \geq 0$, $w[j...] \models \phi$

LTL Models. Each LTL formula provides a set of models. Given a formula ϕ, its set of models can be defined as:

$$L(\phi) = \{\ w \in \Sigma^\omega \mid w \models \phi\ \}$$

This leads to the notion of *equivalence* between two LTL-formulae. Let ϕ and ψ be two LTL-formulae, they are equivalent iff $L(\phi) = L(\psi)$ and we write:

$$\phi \equiv \psi$$

Limitations of LTL for Runtime Verification. Let us consider the following sequence of states observed so far, where ϕ and ψ are different formulae:

$$\phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi$$

Does the sequence satisfy $\Diamond\psi$? Given that the sequence is finite, an answer based on Pnueli's semantics, which takes infinite sequences or paths into account, is hard to provide: $\Diamond\psi$ is in fact satisfied if, from some time point $j \geq 0$ on, ψ becomes true. If the sequence is only a prefix of a (possibly) longer sequence, where the next states are unknown so far just because they must still be observed, then it might be the case that $\Diamond\psi$, if in some successive observed state ψ became true. Given that runtime verification aims monitoring the behaviour of a system, and raising errors only when these errors actually took place, the correct answer to the question above *in a runtime verification setting* would be *it might, or it might not...*. In other words, the verdict is *inconclusive*.

LTL$_3$ has been proposed by Bauer, Leucker, Schallhart [7] in order to make LTL suitable for runtime verification. As discussed in the sequel, LTL$_3$ is defined on finite traces and its semantics is different from the LTL one since three truth values are used: *true*, $false$ and *inconclusive*.

2.2 Three-Valued LTL

Three-Valued Semantics Introduction. Let w be a finite word and ϕ a property. We can distinguish three situations according to what we can prove from w:

- ϕ holds always, even if we do not know the behaviour of the system in the future. We can evaluate ϕ on the finite word w to $true$ ($\top$).
- ϕ will not hold in any scenario. We can evaluate ϕ on the finite word w to $false$ ($\bot$).
- Neither $true$ nor $false$ values can be determined for ϕ. We say that ϕ on the finite word w is *inconclusive* (?)

Monitors. Given the LTL_3 logics, we can define runtime monitors that evaluate finite portions (*prefixes*) of infinite traces. The next elements of a given prefix are called *continuations*; the three-valued semantics can be formally defined in terms of prefixes and continuations. A monitor able to evaluate a given LTL_3 formula on a finite prefix can be implemented as a Finite State Machine with only three output symbols that correspond to the three truth values. The prefixes will be evaluated as *good* or *bad* leading respectively to $true$ and $false$, in the other cases they will be considered *inconclusive*.

2.3 Real-Time Setting

Runtime verification can be applied to systems that emit events at specific time points. These *event-triggered* systems are characterized by time-stamps associated with the events. A run of such a system leads to a *timed word* where each element belongs to Σ x $\mathbb{R}^{\geq 0}$. In order to cope with timed words, we need a logic whose semantics is able to express their properties. We will consider Metric Temporal Logic as an example of Timed Linear Temporal Logic [17] which is a variant of LTL.

Timed Words. We can write a formal definition for timed words following from [1]. A timed word w over the alphabet Σ is a sequence $(a_0, t_0)(a_1, t_1)...$ where each (a_i, t_i) is a *timed event* $\in \Sigma$ x $\mathbb{R}^{\geq 0}$ such that:

- $\forall i \in \mathbb{N}$ we have that $t_i < t_{i+1}$ (*strict monotonicity*)
- if w is infinite, $\forall t \in \mathbb{R}^{\geq 0}$ $\exists i \in \mathbb{N}$ with $t_i > t$ (*progress*)

Starting from the considerations we made in the previous sections, we need to deal with infinite words. In case of a finite timed word $w = (a_0, t_0)(a_1, t_1)...(a_i, t_i)$, its *continuations* are the timed words that start with (a_{i+1}, t_{i+1}) such that $t_{i+1} > t_i$

Where. Timed logics are useful every time the crucial point of the analysis is the time. For example we can consider a distributed, *asynchronous* system in which we want to model not only the single elements but also the whole system.

2.4 Metric Temporal Logic

In this section we introduce Metric Temporal Logic (MTL) and Metric Interval Temporal Logic (MITL) as examples of timed logics. Many logics of this kind exist but we selected these since they are the simplest timed counterparts of LTL. As we will see, LTL and MTL share many underlying notions.

MTL Syntax. Let AP be a set of atomic propositions, MTL formulae are built as:

$$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\, \phi \mid \phi_1 \bigcup_I \phi_2$$

Where $a \in AP$ and $\bigcup$ is the *until* operator we saw before.

We can see that MTL syntax is very similar to the LTL one. The big difference lies inside I which is a an interval that can be open, closed or half open. We have that $I \subseteq \mathbb{R}_{\geq\, 0}$ whose left and right arguments belong to $\mathbb{N} \cup \{\infty\}$. If we consider:

$$\phi ::= \phi_1 \bigcup \phi_2$$

we assume that $I = [0,\, \infty)$ which is the case of LTL formulae. Also in this case we can derive the usual *next*, *always* and *eventually* operators constrained by the intervals:

- $\bigcirc_I\ \phi \equiv \bot \bigcup_I \phi$
- $\mathbf{F}_I\phi = \lozenge_I\ \phi \equiv \top \bigcup_I \phi$
- $\mathbf{G}_I\phi = \square_I\ \phi \equiv \neg\lozenge_I\ \neg\phi$

MTL Semantics. MTL semantics can be *point-based* or *continuous*. The first one is applied when we deal with timed events (which is our case of study) while the other is applied considering signals.

We have to recall LTL Semantics and Timed Words. We report only the semantics for the *until* operator for the sake of simplicity since the semantics is very similar to the LTL one.
Let $\rho = (\sigma,\, \tau)$ be a *timed word* (where $\sigma = \sigma_1\sigma_2...$ is a non-empty finite or infinite word and $\tau = \tau_1\tau_2...$ is a time sequence of the same length such that each couple (σ_i, τ_i) is a timed event) and ϕ a MTL formula. The *satisfaction* relation $\rho \models \phi$ is defined as (for *until* operator):

- $\rho \models \phi_1 \bigcup_I \phi_2$ iff $\exists j$ such that $0 < j < |\rho|$, $\rho[j...] \models \phi_2$, $\rho[k...] \models \phi_1$ $\forall k$ such that $0 < k < j$ and $(\tau_j$ - $\tau_0) \in I$

MTL Models. Each MTL formula defines a set of models that can be classified according to their length. Let ϕ be a MTL formula. The set of *finite* models is defined as:

$$L_f(\phi) = \{\rho \in (\Sigma \text{ x } \mathbb{R}^{\geq\, 0}) : \rho \models \phi\}$$

The set of *infinite* models is defined in the same way and is denoted by $L_\omega(\phi)$.

Recalling the properties stated in Section 2.3, the difference between the two sets is that all the infinite words satisfy both *monotonicity* and *progress* while the finite ones only *monotonicity.*

2.5 Metric Interval Temporal Logic

Metric Interval Temporal Logic shares both the syntax and the semantics with MTL so we can say that it is a sort of restricted MTL. The difference is that we add the constraint on time intervals, in particular to what is called *punctuality.* Let $\bigcup_I$ be the *until* operator and $I = [a,\, b]$ with $a,\, b \in \mathbb{R}_{\geq\, 0}$. We impose that $b > a$ so we cannot have $I = [a,\, a]$. MITL [2] was introduced since MTL models cannot be translated into automata, as we did for LTL. In this case we can introduce Timed Büchi Automata in which we have to add time constraints.

Timed Büchi Automaton. A Timed Büchi Automaton, TBA, in an extension of [9] in which we add *clock constraints.*

Definition (Clock Constraints) A *clock constraint* is a conjunctive formula

$$x \bowtie a$$

Where:

- x is a clock
- a is a constant
- $\bowtie \in \{<,\, >,\, \leq,\, \geq\}$

More formally, a TBA is a tuple $(S,\, S_0,\, X,\, I,\, E,\, F,\, AP,\, L)$ where:

- S is a set of states
- S_0 is a set of initial states such that $S_0 \subseteq S$
- X is a set of clocks
- I: $S \rightarrow \phi_X$ is a map that labels states into sets of clock constraints
- $E \subseteq S$ x ϕ_X x 2^X x S is a set of transitions
- $F \subset S$ is the set of final states
- AP is the set of atomic propositions
- L is a function that labels each state with a subset of AP

So a state is a pair $(s,\, v)$ with $s \in S$ and v is a clock valuation that satisfies the constraints $I(s)$. In order to change state a timed event has to satisfy at least one clock constraint defined in the set of transitions E; the transition leads to a new state that satisfies its set of constraints specified by I. We can recall the concept of *accepting* run we saw before.

Where. MITL is used for model checking. Given a multi-agent system, we can model each agent with *timed runs.* We can also model the whole system through a *collective run* [13].

3 Timed Trace Expressions

In this section we discuss how to extend Trace Expressions with time intervals. We first summarize the Trace Expressions formalism, and then we present its extension, along with examples.

3.1 Trace Expressions

Trace expressions [3,4,5] are based on the notions of *event* and *event type*. We denote by $\mathcal{E}$ the fixed universe of events subject to monitoring. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$, and a trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$. Trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), each specifying a subset of events in $\mathcal{E}$. A trace expression $\tau \in \mathcal{T}$ represents a set of possibly infinite event traces, and is defined on top of the following operators, some of which have already been introduced in Section 1:

- ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ.
- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event ev matches the event type ϑ, and the remaining part is a trace of τ.
- $\tau_1 {\cdot} \tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.
- $\tau_1 {\wedge} \tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2.
- $\tau_1 {\vee} \tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2.
- $\tau_1 | \tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces of τ_1 with the traces of τ_2.

Trace expressions support recursion through cyclic terms expressed by finite sets of recursive syntactic equations, as supported by modern Prolog systems.

The semantics of trace expressions is specified by a transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ and $\mathcal{E}$ denote the set of trace expressions and of events, respectively. We write $\tau_1 \xrightarrow{ev} \tau_2$ to mean $(\tau_1, ev, \tau_2) \in \delta$; the transition $\tau_1 \xrightarrow{ev} \tau_2$ expresses the property that the system under monitoring can safely move from the state specified by τ_1 into the state specified by τ_2 when event ev is observed. A trace expression models the current state of a protocol. Protocol state transitions are ruled by the transition system shown in Figure 1, which define δ.

3.2 Timed Trace Expressions

In order to constrain the set of event traces denoted by a trace expression we need a set of *time constraints*.

Timed Events. In Timed Trace Expressions, events are associated with the time when they have been observed. A timed event is represented by a couple $\langle ev, t\rangle$, where ev is the observed event, and t is the time stamp associated with it.

$$\text{(prefix)}\frac{}{\vartheta{:}\tau \xrightarrow{ev} \tau}\ ev{\in}\vartheta \qquad \text{(or-l)}\frac{\tau_1 \xrightarrow{ev} \tau_1'}{\tau_1{\vee}\tau_2 \xrightarrow{ev} \tau_1'} \qquad \text{(or-r)}\frac{\tau_2 \xrightarrow{ev} \tau_2'}{\tau_1{\vee}\tau_2 \xrightarrow{ev} \tau_2'}$$

$$\text{(and)}\frac{\tau_1 \xrightarrow{ev} \tau_1' \quad \tau_2 \xrightarrow{ev} \tau_2'}{\tau_1{\wedge}\tau_2 \xrightarrow{ev} \tau_1'{\wedge}\tau_2'} \qquad \text{(shuffle-l)}\frac{\tau_1 \xrightarrow{ev} \tau_1'}{\tau_1|\tau_2 \xrightarrow{ev} \tau_1'|\tau_2} \qquad \text{(shuffle-r)}\frac{\tau_2 \xrightarrow{ev} \tau_2'}{\tau_1|\tau_2 \xrightarrow{ev} \tau_1|\tau_2'}$$

$$\text{(cat-l)}\frac{\tau_1 \xrightarrow{ev} \tau_1'}{\tau_1{\cdot}\tau_2 \xrightarrow{ev} \tau_1'{\cdot}\tau_2} \qquad \text{(cat-r)}\frac{\tau_2 \xrightarrow{ev} \tau_2'}{\tau_1{\cdot}\tau_2 \xrightarrow{ev} \tau_2'}\ \epsilon(\tau_1)$$

Figure 1. Operational semantics of trace expressions

Timed Event Types. Time constraints must be also associated with event types which become couples as well:

$$\vartheta = \langle\ \xi_\vartheta,\ \mathcal{C}_\vartheta\ \rangle$$

where the first element denotes a set of events while the second one is a conjunction of time intervals, identifying all the time instants where an event $ev \in \xi_\vartheta$ can be observed, to match the event type ϑ.

With respect to trace expressions, the only modification in timed trace expressions lies in the notions of events and event types, and in the definition of when an event belongs to an event type, which is given pairwise on the two elements of the couples: $\langle\ ev,\ t\ \rangle \in \langle\ \xi_\vartheta,\ \mathcal{C}_\vartheta\ \rangle$ iff $ev \in \xi_\vartheta$ and $t \in \mathcal{C}_\vartheta$.

By default, any event type is defined in the interval $[0,\ \infty)$. This provides a means to transform each trace expression into a timed one, just by associating this interval to each event type.

The semantics is the same as that of trace expressions.

This extension is extremely simple, and if fact the implementation of a "timed monitor driven by a timed trace expression" comes almost for free, by extending the existing Prolog implementation of the "non-timed" monitor. This simplicity has one major drawback described by Examples 1 and 2 below.

Example 1. Let us assume that we have an event a and we intend to model the situation where a should take place two times, the first one in the interval $[0, 5)$ and the second one in the interval $[10, 20]$. If we write a timed trace expression in this way:

$$A = \langle\{a\},\ [0, 5) \cup [10, 20]\rangle$$

$$\tau = A : A : \epsilon$$

τ does not correctly model the scenario we have in mind, as the sequence $\langle a, 1\rangle$ $\langle a, 2\rangle$ respects the formal specification, but not our intuition. We point out that an event can belong to different timed event types; this leads to the following solution:

$$A = \langle \{a\}, [0, 5) \rangle$$
$$B = \langle \{a\}, [10, 20] \rangle$$

$$\tau = A : B : \epsilon$$

Example 2. We consider a more sophisticated example, where the monitor can accept an event a in different time intervals, and according to the timestamp of the event the execution proceeds on a different branch, for example by moving to τ_1 in one case, and by waiting for an event b in the second case, and then moving to τ_2. The resulting timed trace expression might look like:

$$A = \langle \{a\}, [2, 7) \cup (15, 20] \rangle$$
$$B = \langle \{b\}, [0, \infty) \rangle$$

$$\tau = A : \tau_1 \lor A : B : \tau_2$$
$$\tau_1 = \textit{...something...}$$
$$\tau_2 = \textit{...something...}$$

But this definition of τ is nondeterministic, as $\langle a, 4 \rangle$, as well as $\langle a, 17 \rangle$ might both be accepted to move either in τ_1 or in $B : \tau_2$. We must rewrite the trace expressions as follows, by creating a new timed event type:

$$A = \langle \{a\}, [2, 7) \rangle$$
$$B = \langle \{b\}, [0, \infty) \rangle$$
$$C = \langle \{a\}, (15, 20] \rangle$$

$$\tau = A : \tau_1 \lor C : B : \tau_2$$
$$\tau_1 = \textit{...something...}$$
$$\tau_2 = \textit{...something...}$$

In this case, if the monitor observes the event a with timestamp $t \geq 16$, the rightmost branch must be selected to continue the monitoring.

Example 3. Finally, we can note that a trace expression τ might be "unsatisfiable", in the sense that no actual trace of timed events can meet the specification given by τ. The following simple trace exemplifies this situation.

$$A = \langle \{a\}, [5, 9) \rangle$$
$$B = \langle \{b\}, [0, 3) \rangle$$

$$\tau = A : B : \epsilon$$

In this trace, the temporal constraints are not consistent with respect to the properties of *timed words* such as *monotonicity*; the : operator forces a to take place before b, but the intervals associated with A and B make this structural property of τ not satisfiable.

Discussion: limitations of Timed Trace Expressions. Examples 1 and 2 show that the simple formalization that may come to the mind of the trace expression designer to meet some informal specification, may easily turn out to be the wrong one. The burden of ensuring that different instances of the same event that must take place in different intervals are modelled by distinct event types, insists entirely on the designer's shoulders. When timed trace expressions are more complex than those shown in this paper, and when they are defined in a recursive way, the correct formalization may be difficult to specify. One possible solution to overcome this problem, could be to associate intervals with sub-traces, rather that with event types. In fact, intervals associated with event types have a global scope, but in some cases it might be more convenient to have intervals with a local scope. This extension, however, would require to change the syntax of trace expressions (while so far we only changed the syntax of events and event types), by introducing an explicit notion of "scope of an interval", as we did with parametric trace expressions [5] for the notion of "scope of a parameter". This would also require a change in the semantics.

The other feature that we point out, is the ability to write timed trace expressions which are useless, as their structure is non compliant with the temporal constraints therein. Given that the monitoring engine for timed trace expressions is implemented in SWI Prolog[4], we are exploring the possibility to exploit the support that SWI Prolog offers to constraint logic programming[5], to detect these design errors "at compile time".

4 Conclusions and Future Work

Different techniques exist for definition, model checking and runtime verification of complex, distributed systems where time plays a crucial role, such as Temporal Logics. Interval Temporal Logic is one of the first formalisms that had an impact in the Multiagent System research field even if it was born for hardware reasoning. As the name suggests, the time is specified in terms of intervals. On the other hand, Linear Temporal Logic is a kind of modal logic that can be used for specifying temporal properties of the systems. It was proposed for formal verification by Amir Pnueli [15] and it was extended to LTL_3 by Bauer, Leucker, Schallhart [7] for runtime verification.

In real-time systems it is likely that time bounded properties must be specified: Timed Linear Temporal Logic was proposed by Raskin [17] and it is the counterpart of LTL for timed words. As for LTL, Timed LTL can be extended for runtime verification (TLTL_3, [7]). Metric Temporal Logic is another example of Timed Logic: a subset of MTL, which is called Metric Interval Temporal Logic, is usually considered due to its good decidability properties.

In this paper we have applied the ideas supported by the logics above to the Trace Expressions formalism, resulting into Timed Trace Expressions.

[4] http://www.swi-prolog.org/, accessed on March 25, 2019.

[5] http://www.swi-prolog.org/pldoc/man?section=clp, accessed on March 25, 2019.

Our future directions of research are related with overcoming the limitations discussed in Section 3. Besides this, in [4] we demonstrated that an LTL formula (with LTL_3 semantics) can be translated into an equivalent Trace Expression passing through the concept of Büchi Automaton. We aim at investigating if the same procedure can be applied to TLTL_3 and Timed Trace Expressions through Timed Büchi Automata.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. In L. Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 139–152. ACM, 1991.
3. D. Ancona, A. Ferrando, L. Franceschini, and V. Mascardi. Parametric trace expressions for runtime verification of Java-like programs. In *FTfJP@ECOOP*, pages 10:1–10:6. ACM, 2017.
4. D. Ancona, A. Ferrando, and V. Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In *Theory and Practice of Formal Methods*, volume 9660 of *LNCS*, pages 47–64, 2016.
5. D. Ancona, A. Ferrando, and V. Mascardi. Parametric runtime verification of multiagent systems. In K. Larson, M. Winikoff, S. Das, and E. H. Durfee, editors, *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pages 1457–1459. ACM, 2017.
6. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.
7. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.
8. P. Bellini. Interval temporal logic for real-time systems: Specification, execution and verification processes. PhD. Thesis, University of Florence, Italy, 2001.
9. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*. Stanford University Press, 1962.
10. T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer, 1991.
11. N. R. Jennings, K. P. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
12. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
13. A. Nikou, J. Tumova, and D. V. Dimarogonas. Cooperative task planning of multi-agent systems under timed temporal specifications. In *2016 American Control Conference, ACC 2016, Boston, MA, USA, July 6-8, 2016*, pages 7104–7109. IEEE, 2016.

14. J. Ouaknine and J. Worrell. Some recent results in metric temporal logic. In F. Cassez and C. Jard, editors, *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, volume 5215 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2008.
15. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Comp. Soc. Press, 1977.
16. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency, Overviews and Tutorials*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer, 1986.
17. J.-F. Raskin. Logics, automata and classical theories for deciding real-time. PhD. Thesis, Facultés universitaires Notre-Dame de la Paix, Namur, 1999.
18. F. Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, 2004.
19. M. Wooldridge and N. R. Jennings, editors. *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 8-9, 1994, Proceedings*, volume 890 of *Lecture Notes in Computer Science*. Springer, 1995.
20. M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *Knowledge Eng. Review*, 10(2):115–152, 1995.