An Event-driven Interval Temporal Logic for Hybrid Systems* (Work in Progress)

María-del-Mar Gallardo and Laura Panizo
Universidad de Málaga, Andalucía Tech,
Departamento de Lenguajes y Ciencias de la Computación,
Campus de Teatinos s/n, 29071, Málaga, Spain
{gallardo, laurapanizo}@lcc.uma.es

Nowadays, hybrid systems are present in many crucial tasks of our daily life. The hybrid character derives from the merge of continuous and discrete dynamics that are intrinsically related. The verification of critical properties of hybrid systems is of special importance, but sometimes it is not feasible due to their inherent complexity. In the last few years, several model-based testing and runtime verification techniques have been proposed to support the verification and validation of hybrid systems. In this paper, we present an interval logic that is suitable for specifying properties of event-driven hybrid systems. We introduce the syntax and semantics of the logic, and propose an automatic mechanism to transform each formula in the logic into a network of timed automata that can act as observers of the property in each test case using the UPPAAL tool.

1 Introduction

Hybrid systems are characterised by the combination of the so-called *continuous* and *discrete behaviours*. The continuous component is typically composed by real-valued variables that describe the physical real world and whose values evolve over time. Usually, the discrete component interacts with the continuous one to control and update, when necessary, the evolution of the continuous variables. In the last few years, principally due to the improvement in sensor technology, many new hybrid (also called cyber-physical) systems supporting different tasks have appeared. For instance, hybrid systems may be used to help independent elderly people through in-home monitoring, checking their movements to fire an alarm in the case an anomalous behaviour is detected [6]. In aeronautics, for example, hybrid systems can be used to detect if two drones are too close to each other [13]. Finally, in the health field there exist many applications of hybrid systems. An interesting case corresponds to the development of an artificial pancreas to help diabetic people control their blood sugar levels [8].

These examples show that the tasks carried out by hybrid systems are, in most cases, critical and therefore, it is necessary to guarantee that the systems behave correctly, at least, with respect to their essential properties. Hybrid systems may be described as *hybrid automata* where continuous variables are allowed to evolve in automata locations (discrete states) following dynamics given by differential equations. Verification by *model checking* of hybrid automata is, in general, undecidable except for some well-known types of automata. Existing tools, such as UPPAAL [4] or PHAVER [12], focus on the verification of hybrid automata subclasses. UPPAAL analyses systems described *as timed automata* in which all continuous variables are clocks evolving at the same rate. PHAVER performs reachability analysis in systems described as *linear hybrid automata* in which continuous dynamics may be linear differential equations.

*This work has been supported by the Spanish Ministry of Economy and Competitiveness project TIN2015-67083-R and the European Unions Horizon 2020 research and innovation programme under grant agreement No 688712

Submitted to:
PROLE 2018

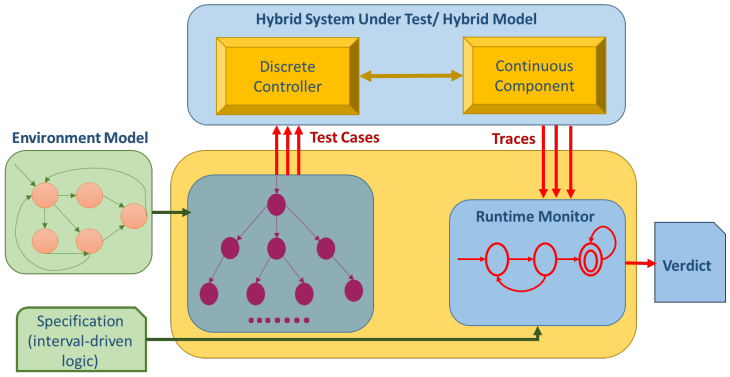


Figure 1: Model-based Testing & Runtime Verification approaches

Although these tools are truly useful for analysing hybrid systems, they do not completely solve the problem. On the one hand, some continuous variables follow complex dynamics that are hard to model. It is, of course, possible to apply techniques such as *abstract interpretation* to approximate the state space, but not all continuous variables can be effectively approximated using this technique. On the other hand, hybrid systems usually interact with an open environment whose behaviour is, a priori, unknown. As a result, in the last few decades, new hybrid computation models have appeared to capture these characteristics. For instance, *extended hybrid systems* [1, 11] allow the evolution of continuous variables, which may also depend on new parameters representing the influence of the environment. Sampled-data control systems [15] constitute a different model of hybrid systems where continuous transitions have a fixed duration and they are followed by discrete transitions that take place each predefined time instant.

Approaches for the analysis of complex hybrid systems combining *model-based testing* [7] and *runtime verification* [16, 14] techniques are currently gaining attention from researchers and industry. Figure 1 shows a possible model of this interaction. Model-based testing aims to automatically produce test cases based on a behavioural model of the system under test (SUT). In most cases, the SUT can be instrumented to produce execution traces that can be observed by means of runtime verification techniques to determine whether the desired properties hold.

In this context, the language for properties is a key component since, on the one hand, it must be expressive enough to describe properties of interest and, on the other, it must be possible to transform specifications into monitors that analyse the correctness of traces. In this paper, we focus on the problem of how to describe critical properties of hybrid systems using a version of the *interval-based logics*. The main actors of *interval-based logics* are time intervals that represent a continuous time space where the system has to fulfil some constraints. Instead of describing intervals using real-valued constants, our proposal is to define an extension of the future LTL logic that uses events to determine the time intervals where the continuous variables must be monitored. For instance, assume that one possible behaviour of the patient with the artificial pancreas is to go running from time to time. The exact moment when the patient starts to run is a priori unknown, but once he/she has started running, it is very important to check that his/her sugar level never drops below 90 mg/l. Thus, the time during which the value of the sugar levels must be monitored corresponds with the trace states when events `start_running` and `end_running` take place. Of course, there are many other LTL interval logics which will be discussed in Section 2. However, the proposal is intended to improve/adapt these approaches to the context of model-based techniques, providing sufficient expressiveness to describe as many properties as possible,

while at the same time we maintain logic decidability. To this end, we present the translation of logic specifications into *deterministic timed automata* which will behave as observers of the execution traces.

The paper is organised as follows. Section 2 summarises some work related to interval logics. Section 3 presents the syntax and semantics of the event-driven interval logic. Section 4 describes a methodology to transform each interval formula into a network of timed automata, which allows us to check the satisfaction of the formula using the tool UPPAAL. We also present some case studies and, finally, Section 5 gives the conclusions and future work.

2 Related Work

Linear Temporal Logic (LTL) has been extensively used to describe requirements on concurrent and distributed systems. Although it has sufficient expressive power to model many desired properties, LTL also shows some weaknesses. For instance, to order several events over time, one has to use some nested *until* operators which may make the formula hard to manage. In addition, LTL semantics of modal operators $\Box$, $\Diamond$ and *until* are thought to express requirements in an unbounded future; this is why it is difficult to specify properties that have to be true before a certain deadline occurs. Thus, the extension of LTL with *intervals* seems a natural idea to make it easier to express these type of properties. This is the approach followed in [22], where the authors use events to determine the intervals on which formulae must be evaluated, although they do not deal with real-time. The temporal logic FIL [20] was also defined with similar purposes but using formulae written using a graphical representation to describe properties. Real-time FIL [21] is an extension of FIL that incorporates a new predicate $len(d_1, d_2]$ that bounds the length of the intervals on which properties have to be evaluated.

The duration calculus [10] (DC) was defined to verify real-time systems. The main contribution of DC is to consider that system states have a *duration* in each time interval which may be measured taking into account the presence of the state in the interval. DC includes modalities (temporal operators) able to express relations between intervals and states which constitute the basis of the logic. For instance, the formula $\Box((\llbracket Leak \rrbracket \frown \llbracket \neg Leak \rrbracket \frown \llbracket Leak \rrbracket) \Rightarrow l \geq 30)$ is satisfied in a gas burner system if a non leaking state, with a duration greater than 30 time units, always exists between two leak states. The operator $\frown$ represents that intervals must be consecutive.

Metric Interval Logic (MITL) [3] is a real-time temporal logic that extends LTL by using modal operators of the form $\Box_I$, $\Diamond_I$ where I is an open/close, bounded/unbounded interval of $\mathbb{R}$. Thus, it is possible to write formulae such as $\Box_{\geq 0}(p \rightarrow \Diamond_{(0,3]} q)$ to specify the property "*every p-state is followed by a q-state within 3 time units*". The logic $\text{MITL}_{[a,b]}$ [18] was defined as a bounded version of MITL, such that all temporal modalities are restricted to intervals of the form $[a,b]$ with $0 \leq a < b$ and $a, b \in \mathbb{N}$. $\text{MITL}_{[a,b]}$ has future and past temporal operators and its formulae can be translated into deterministic timed automata.

More recently, $\text{MITL}_{[a,b]}$ was extended to signal temporal logic STL [17] including numerical predicates that allow analog and mixed-signal properties to be specified. For instance, in STL, it is possible to express the property "*The absolute value of a continuous signal x is always less than* 6 *and when the (Boolean) trigger rises, within 600 time units* $|x|$ *has to drop below* 1 *and stay like that for at least* 300 *time units*" as $\Box(|x| < 6 \wedge (trigger \rightarrow \Diamond_{[0,600]} \Box_{[0,300]} (|x| < 1)))$. Observe that, in this formula, Boolean expressions $|x| < 6$ and $|x| < 1$ refer to the value of the continuous variable x within intervals $[0,600]$ and $[0,300]$. More recently, this logic has been extended to XSTL [19] by adding *timed regular expressions* to express behaviour patterns to be met by signals.

3 Formalisation of Logic

In this section, we first introduce a very general model of hybrid system with the aim of representing as many hybrid systems as possible. Then, we introduce the syntax and semantics of the event-driven interval temporal logic, and illustrate its utility with some examples.

3.1 Model of a Hybrid System

We assume that the system behaviour is given by a timed transition system $P = \langle \Sigma, \overset{-}{\longmapsto}_d, \overset{-}{\longmapsto}_t, \mathcal{E}, s_0 \rangle$ where Σ is a non-numerable set of observable states, $\mathcal{E}$ is a finite set of labels, $\overset{-}{\longmapsto}_d, \overset{\iota}{\longmapsto}_t \subseteq \Sigma \times \mathcal{E} \times \Sigma$ are the discrete and continuous transition relations ($t \in \mathbb{R}^{\geq 0}$ being the duration of the continuous transition, and $\iota \in \mathcal{E}$ representing an *internal* continuous transition), and $s_0 \in \Sigma$ is the initial state. We denote with $\mathcal{O}_f(P)$ the set of execution traces of finite length determined by P. The elements of $\mathcal{O}_f(P)$ are traces of the form $\pi = s_0 \overset{e_0}{\longmapsto}_{\lambda_0} s_1 \overset{e_1}{\longmapsto}_{\lambda_1} \cdots \overset{e_{n-1}}{\longmapsto}_{\lambda_{n-1}} s_n$ where each $\lambda_i \in \{d\} \bigcup \mathbb{R}^{\geq 0}$ denotes the type of the transition (discrete or continuous), and $e_i \in \mathcal{E}$ is the event that fired the transition. Given a trace $\pi = s_0 \overset{e_0}{\longmapsto}_{\lambda_0} s_1 \overset{e_1}{\longmapsto}_{\lambda_1} \cdots \overset{e_{n-1}}{\longmapsto}_{\lambda_{n-1}} s_n$, we define the set $\mathcal{O}(\pi)$ as the set of observable states of π, that is, $\mathcal{O}(\pi) = \{s_0, \cdots, s_n\}$.

Note that the previous definition is highly general in the sense that it is able to capture the behaviour of very different hybrid systems, for instance, those that are described by hybrid automata or by means of other different formalisms. Discrete transitions correspond to discrete changes of the system variables and *always produce observable states* in the traces. In contraposition, continuous transitions model the evolution of continuous variables, and the changes of these variables are only visible at some time instants. Thus, in the following discussion, we assume that the evolution of continuous variables between two observable states may be known or approximated. For instance, in the case that the traces are taken from initialised rectangular hybrid automata, the values of continuous variables during continuous transitions are completely known. For other models, it is usual to assume that the differential equations guiding the changes of continuous variables satisfy some stability requirements in such a way that the unknown values of continuous variables can be approximated.

Assuming that the initial state s_0 of each trace $\pi \in \mathcal{O}_f(P)$ happens at time instant 0, the duration of a trace $\delta(\pi) \in \mathbb{R}^{\geq 0}$ is the time instant when the last state of π occurs. Thus, given a trace π, we define function $\tau : \mathcal{O}(\pi) \mapsto [0, \delta(\pi)]$ that associates each observable state of π with the time instant where it occurs, that is, $\tau(s_i)$ is the time when s_i took place. We denote with $\mathcal{T}(\pi) = \{t_0, \cdots, t_n\}$ the time instants where the behaviour of π is observable. Observe that τ is a bijection between $\mathcal{O}(\pi)$ and $\mathcal{T}(\pi)$. We call $\sigma : \mathcal{T}(\pi) \to \mathcal{O}(\pi)$ the inverse function of τ, i.e., $\tau(\sigma(t_i)) = t_i$ and $\sigma(\tau(s_i)) = s_i$. Considering this, given a trace π and $t \in \mathcal{T}(\pi)$, we denote with $\langle \pi, t \rangle$ the observable state of the trace at time instant t.

3.2 Event-driven LTL

We consider two types of (atomic) propositions to be analysed on traces of $\mathcal{O}_f(P)$. On the one hand, we have those that refer to single states of traces as used in propositional temporal logic LTL, for instance. To this end, let $\mathcal{F}$ be a set of state formulae to be evaluated on states of Σ. Relation $\vdash \subseteq \Sigma \times \mathcal{F}$ associates each state with the state formulae it satisfies, that is, given $s \in \Sigma$, and $p \in \mathcal{F}$, $s \vdash p$ iff the state s satisfies the state formula p. As usual, we assume that state formulae are constructed from a set of atomic propositions and Boolean operators. In the following, given $\pi \in \mathcal{O}_f(P)$, $t_i \in \mathcal{T}(\pi)$ and $p \in \mathcal{F}$ we write $\langle \pi, t_i \rangle \vdash p$ iff $\sigma(t_i) = s_i \vdash p$. To simplify the formalisation, we assume that the labels of discrete transitions introduced in Section 3.1 represent events and that states are able to reflect whether an event has been fired.

In order to analyse the behaviour of continuous variables, it is useful to observe them not only in a given time instant, but also during time intervals to know, for example, whether their values hold inside some expected limits. To this end, we use intervals of states (inside the traces) to determine the periods of time during which continuous variables should be observed. To do this, we are inspired by the interval calculus introduced by [10] where the domain of interval logic is the set of time intervals $\mathbb{I}$ defined as $\{[t_1,t_2] | t_1,t_2 \in \mathbb{R}, t_1 \leq t_2\}$. Considering this, we define the so-called *interval formulae* as functions of the type $\phi : \mathbb{I} \to \{true, false\}$ to represent the atomic formulae of the interval logic that represent the expected behaviour of continuous variables. For instance, assume that $c : \mathbb{R}^{\geq 0} \to \mathbb{R}$ is a continuous variable of our system, $c(t)$ being the value of c at time instant t. Given a constant, K, formula $\phi_c : \mathbb{I} \to \{true, false\}$ given as $\phi_c([t_1,t_2]) = |c(t_2) - c(t_1)| \leq K$ defines an interval formula that is *true* on an interval $[t_1,t_2]$ iff the difference between the value of c in the interval endpoints t_1 and t_2 is less than the constant K. Let us denote with Φ a set of interval formulae. We assume that Φ contains the interval formula $\mathfrak{T}true : \mathbb{I} \to \{true, false\}$ which returns *true* for all real intervals, that is, $\forall I \in \mathbb{I}.\mathfrak{T}true(I) = true$.

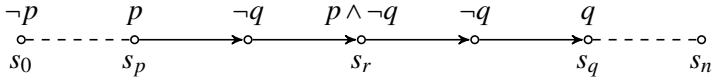
In the following, given two state formulae $p, q \in \mathcal{F}$, we use expressions of the form $[p,q]$, that we call *event intervals* to denote intervals of states in traces. Intuitively, given a trace $\pi = s_0 \overset{e_0}{\longmapsto}_{\lambda_0} s_1 \overset{e_1}{\longmapsto}_{\lambda_1} \cdots \overset{e_{n-1}}{\longmapsto}_{\lambda_{n-1}} s_n$, $[p,q]$ represents time intervals $[t_i,t_j]$ of π such that $\langle \pi, t_i \rangle \vdash p$ and $\langle \pi, t_j \rangle \vdash q$, that is, $s_i \vdash p$ and $s_j \vdash q$. In addition, we use the interval $[p,-]$ to denote simple states in π that satisfy p. Now, we formally define the relation $\Vdash$ that relates event intervals with intervals of states in traces.

Definition 3.1 *Given a trace $\pi \in \mathcal{O}_f(P)$, two time instants $t_p, t_q \in \mathcal{T}(\pi)$ such as $t_p < t_q$, and two state formulae $p, q \in \mathcal{F}$, we say that the time interval $[t_p,t_q]$ satisfies the pair of state formulae (p,q) in the context of the trace π, and we denote it as $\pi \downarrow [t_p,t_q] \Vdash [p,q]$, iff the following conditions hold:*

1. $\langle \pi, t_p \rangle \vdash p$,
2. $\forall t_j \in (t_p,t_q) \cap \mathcal{T}(\pi), \langle \pi, t_j \rangle \nvdash q$,
3. $\langle \pi, t_q \rangle \vdash q$
4. *There exists no interval $[t'_p,t_q] \neq [t_p,t_q]$, verifying conditions 1–3 of this definition, such that $[t_p,t_q] \subset [t'_p,t_q]$.*

That is, $\pi \downarrow [t_p,t_q] \Vdash [p,q]$ iff $\sigma(t_p) = s_p$ satisfies p, $\sigma(t_q) = s_q$ is the first state following s_p that satisfies q. In addition, the last condition ensures that the interval of states is maximal in the sense that it is not possible to find a larger interval ending at s_q satisfying the previous conditions. Observe that, in the previous definition, the time instants t_p and t_q must be different elements of $\mathcal{T}(\pi)$, that is, they must correspond to observable states of π and the interval $[t_p,t_q]$ cannot be a point.

Example 3.2 The following trace (π) tries to clarify Definition 3.1. Given state formulae p and q, and assuming that $\tau(s_i) = t_i$ for all states, we have that $\pi \downarrow [t_p,t_q] \Vdash [p,q]$, but $\pi \downarrow [t_r,t_q] \nVdash [p,q]$, since condition (4) does not hold.



■

The following definition gives the syntax of the new event-driven LTL.

Definition 3.3 *[Event-driven LTL formulae] Given two state formulae $p, q \in \mathcal{F}$, and an interval formula $\phi \in \Phi$, the formulae of the* event-driven LTL logic *are recursively constructed as follows:*

$$\psi ::= \phi \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \mathcal{U}_{[p,q]} \psi_2 \mid \psi_1 \mathcal{U}_{[p,-]} \psi_2$$

The rest of the temporal operators are accordingly defined as:

$$\Diamond_{[p,q]}\,\psi \equiv \mathfrak{T}true\,\mathcal{U}_{[p,q]}\psi \qquad \Box_{[p,q]}\psi \equiv \neg(\Diamond_{[p,q]}\neg\psi)$$
$$\Diamond_{[p,-]}\,\psi \equiv \mathfrak{T}true\,\mathcal{U}_{[p,-]}\psi \qquad \Box_{[p,-]}\psi \equiv \neg(\Diamond_{[p,-]}\neg\psi)$$

The following definition gives the semantics of the event-driven LTL formulae given above. Given a trace $\pi \in \mathcal{O}_f(P)$, and $t_i, t_f \in \mathcal{T}(\pi)$ with $t_i \le t_f$, we use the notation $\langle \pi, t_i, t_f\rangle$ to represent the subtrace of π from state $s_i = \sigma(t_i)$ to state $s_f = \sigma(t_f)$.

DEFINITION 3.4 (SEMANTICS OF THE EVENT-DRIVEN LTL FORMULAE) *Given two atomic propositions $p,q \in \mathcal{F}$, an atomic interval formula $\phi \in \Phi$, and the event-driven LTL formulae ψ, ψ_1, ψ_2, the satisfaction relation $\models$ is defined as follows:*

$$\langle \pi, t_i, t_f\rangle \models \phi \quad \textit{iff} \quad \phi([t_i, t_f]) \tag{3.1}$$
$$\langle \pi, t_i, t_f\rangle \models \neg\psi \quad \textit{iff} \quad \langle \pi, t_i, t_f\rangle \not\models \psi \tag{3.2}$$
$$\langle \pi, t_i, t_f\rangle \models \psi_1 \vee \psi_2 \quad \textit{iff} \quad \langle \pi, t_i, t_f\rangle \models \psi_1 \textit{ or } \langle \pi, t_i, t_f\rangle \models \psi_2 \tag{3.3}$$
$$\langle \pi, t_i, t_f\rangle \models \psi_1 \mathcal{U}_{[p,q]} \psi_2 \quad \textit{iff} \quad \exists I = [t_p, t_q] \subseteq [t_i, t_f] \textit{ such that } \pi \downarrow [t_p, t_q] \Vdash [p,q] \textit{ and } \langle \pi, t_i, t_p\rangle \models \psi_1, \langle \pi, t_p, t_q\rangle \models \psi_2 \tag{3.4}$$
$$\langle \pi, t_i, t_f\rangle \models \psi_1 \mathcal{U}_{[p,-]} \psi_2 \quad \textit{iff} \quad \exists t_p.\ t_i \le t_p \le t_f \textit{ and } \langle \pi, t_i, t_p\rangle \models \psi_1, \langle \pi, t_p, t_p\rangle \models \psi_2 \tag{3.5}$$

The semantics given by $\models$ is similar to that of LTL, except that $\models$ manages interval formulae instead of state formulae. For instance, case 3.1 states that the subtrace $\langle \pi, t_i, t_f\rangle$ of π from state s_i to state s_f satisfies an interval formula ϕ iff $\phi([t_i, t_f])$ holds. Maybe, cases 3.4 and 3.5 are more interesting. Definition 3.4 establishes that $\mathcal{U}_{[p,q]}$ holds on the subtrace $\langle \pi, t_i, t_f\rangle$ iff there exists an interval $[t_p, t_q] \subset [t_i, t_f]$ such that ψ_1 and ψ_2 hold on $[t_i, t_p]$ and $[t_p, t_q]$, respectively. Case 3.5 is similar except for the interval in which ψ_2 has to be true is $[t_p, t_p]$, which represents the time instant t_p.

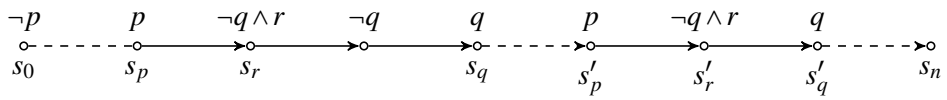
PROPOSITION 3.5 *The semantics of operators $\Box_{[p,q]}, \Diamond_{[p,q]}, \Box_{[p,-]}$ and $\Diamond_{[p,-]}$, given in Definition 3.3, is the following:*

$$\langle \pi, t_i, t_f\rangle \models \Diamond_{[p,q]}\psi \quad \textit{iff } \exists I = [t_p, t_q] \subseteq [t_i, t_f], \textit{ such that } \pi \downarrow [t_p, t_q] \Vdash [p,q] \textit{ and } \langle \pi, t_p, t_q\rangle \models \psi \tag{3.6}$$
$$\langle \pi, t_i, t_f\rangle \models \Box_{[p,q]}\psi \quad \textit{iff } \forall I = [t_p, t_q] \subseteq [t_i, t_f], \textit{ if } \pi \downarrow [t_p, t_q] \Vdash [p,q] \textit{ then } \langle \pi, t_p, t_q\rangle \models \psi \tag{3.7}$$
$$\langle \pi, t_i, t_f\rangle \models \Diamond_{[p,-]}\psi \quad \textit{iff } \exists t_p \in [t_i, t_f] \textit{ such that } \langle \pi, t_p\rangle \vdash p \textit{ and } \langle \pi, t_p, t_p\rangle \models \psi \tag{3.8}$$
$$\langle \pi, t_i, t_f\rangle \models \Box_{[p,-]}\psi \quad \textit{iff } \forall t_p \in [t_i, t_f] \textit{ if } \langle \pi, t_p\rangle \vdash p \textit{ then } \langle \pi, t_p, t_p\rangle \models \psi \tag{3.9}$$

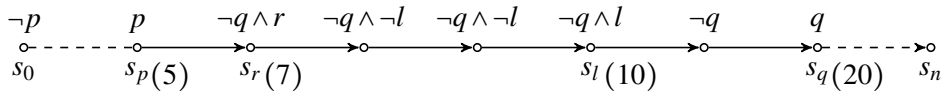
PROOF.
Cases 3.6 and 3.8 follow on from the definition of $\Diamond_{[p,q]}$ and $\Diamond_{[p,-]}$, taking into account that the interval formula $\mathfrak{T}true$ holds for all intervals. Case 3.7. By definition $\Box_{[p,q]}\psi = \neg\Diamond_{[p,q]}\neg\psi$, that is, $\langle \pi, t_i, t_f\rangle \models \Box_{[p,q]}\psi$ iff $\langle \pi, t_i, t_f\rangle \not\models \Diamond_{[p,q]}\neg\psi$. Using expression 3.6, we have that $\langle \pi, t_i, t_f\rangle \not\models \Diamond_{[p,q]}\neg\psi$ iff for all intervals $[t_p, t_q] \subseteq [t_i, t_f]$ such that $\pi \downarrow [t_p, t_q] \Vdash [p,q]$, $\neg\psi([t_p, t_q])$ is *false*, or equivalently, $\psi([t_p, t_q])$ holds. The proof for case 3.9 is similar to the one above.

EXAMPLE 3.6 We now give some examples that show the expressiveness of the logic.

1. The following trace satisfies property the "*Event r occurs at least once between each occurrence of events p and q*" which may be described using the formula $\Box_{[p,q]} \Diamond_{[r,-]} \mathfrak{T}rue$.



2. Given $K \in \mathbb{R}$, let us define the interval formula $\psi_K : \mathbb{I} \to \{true, false\}$ as $\psi_K([t_1,t_2]) = t_2 - t_1 \geq K$. The property "*each time event p occurs, if q occurs in more than* 10 *time instants, then there must occur events r and l in between which are* 3 *instants apart*" maybe described as $\Box_{[p,q]}(\psi_{10} \to \Diamond_{[r,l]} \psi_3)$ and it is satisfied by the following trace.



■

4 Implementation

In this section, we present a prototype implementation of the event-driven interval temporal logic that is based on translating each interval formula into a network of *timed automata* [2, 4] that accepts the traces satisfying the formula. We first introduce the notion of timed automaton and UPPAAL, the tool that supports the implementation.

4.1 Background on Timed Automata

UPPAAL [4, 5] is a model checking tool for verifying real-time systems. Systems are described as a network of timed automata and are verified against a set of properties specified with a subset of TCTL (Timed Computation Tree Logic). The formal model of timed automata was first proposed in [2], but UPPAAL uses an extended definition presented in [4].

A *timed automaton* is a finite-state machine extended with clock variables that evolve linearly, and at the same rate, over time. Timed automata use a dense time model, i.e., clock variables are real-valued variables. In the following definition, C denotes a set of clocks and $B(C)$ is a set of clock constraints of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, >, \geq\}$.

DEFINITION 4.1 (TIMED AUTOMATA) *A timed automaton is a tuple $\langle L, l_0, C, A, E, I \rangle$, where L is a finite set of locations, $l_0 \in L$ is the initial location, C is a set of clocks, A is a set of actions, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ is a function that assigns invariants to locations.*

A clock valuation is a function $u : C \to \mathbb{R}^{\geq 0}$ that gives a non-negative real value to each clock variable. $\mathbb{R}^C$ denotes the set of all clock valuations. Initially, the value of all clocks is 0. By abuse of notation, we consider guards and invariants as sets of clock valuations. Thus, $u \in I(l)$ means that u satisfies the invariant $I(l)$.

A timed automaton state is a pair (l, u), where $l \in L$ is the location of the automaton and $u \in \mathbb{R}^C$ is the valuation of clock variables in C such that $u \in I(l)$. Intuitively, a timed automaton can evolve

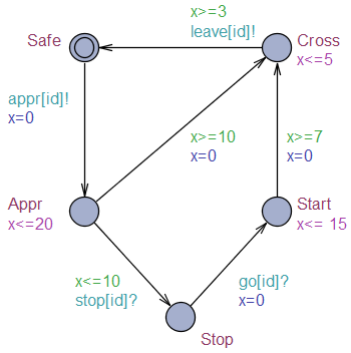


Figure 2: Train automaton

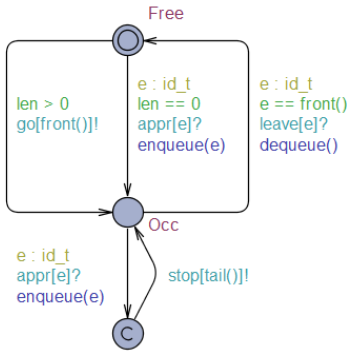


Figure 3: Gate automaton

executing continuous or discrete transitions. During continuous transitions, denoted as $(l,u) \xrightarrow{t} (l,u+t)$, the automaton stays at the same location l, while the clock variables evolve t time units. In contrast, discrete transitions, denoted as $(l,u) \xrightarrow{a} (l',u')$, execute actions and change the automaton location instantaneously, without time passing (some clocks can also be updated).

Definition 4.2 (Semantics of Timed Automata) *Let $T = \langle L, l_0, C, A, E, I\rangle$ be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \xrightarrow{-}\rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\xrightarrow{-} \subseteq S \times \mathbb{R}^{\geq 0} \cup A \times S$ is the transition relation such that:*

- $(l,u) \xrightarrow{t} (l,u+t)$ *if* $\forall t' : 0 \leq t' \leq t . u + t' \in I(l)$, *and*
- $(l,u) \xrightarrow{a} (l',u')$ *if* $\exists e = (l,a,g,r,l') \in E$ *such that* $u \in g, u' = [r \mapsto 0]u$ *and* $u' \in I(l')$,

where for $t \in \mathbb{R}^{\geq 0}$, $u+t$ maps each clock $x \in C$ to the value $u(x)+t$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C - r$.

Example 4.3 Figures 2 and 3 show the automata of the train gate example [4], which model a simple railway control system. The train automaton (Figure 2) has 5 locations (`Safe`, `Appr`, `Stop`, `Start` and `Cross`) and a clock variable (x). In the example, the automaton is initially in location `Safe` (double circled). Locations can have invariants over clock variables (e.g. $x \leq 20$ at location `Appr`) that have hold in order to let time evolve. Similarly, transitions can be guarded by clock constraints that have to be true to enable discrete transitions between two locations. Transitions can also be labelled with synchronisation labels (channels). For instance, a transition labelled with `appr!` synchronises with another labelled with `appr?`. In addition, transitions can be labelled with updates of the clock variables (e.g. $x = 0$) and/or function calls (e.g. `enqueue(e)`). Figure 3 shows the gate controller automaton. The automata has committed location, labelled with a `C`, which is a special type of location defined in Uppaal. Committed locations do not allow time to evolve, and thus the next transition of the automaton has to be an ongoing transition from a committed location.

■

In Uppaal, systems are modelled as networks of timed automata. A network of timed automata is composed by several timed automata $T_i = \langle L_i, l_i^0, C, A, E_i, I_i\rangle$ in parallel, whose clocks evolve synchronously and share a set of synchronisation labels. The state of a network of timed automata is a pair $(\bar{l}, u)$, where $\bar{l} = (l_1, \ldots, l_n)$ is a location vector. The invariant of a location vector is defined as $I(\bar{l}) = \bigwedge_i I_i(l_i)$. To

simplify the notation, we write $\bar{l}[l'_i/l_i]$ to denote the vector where the l_i component is replaced by l'_i. The composition of timed automata in networks is the parallel interleaving semantics with synchronisation, and it is defined below.

DEFINITION 4.4 (SEMANTICS OF A NETWORK OF TIMED AUTOMATA) *Let $T_i = \langle L_i, l_i^0, C, A, E_i, I_i\rangle$ $(i = 0 \dots n)$ be a network of n timed automata, $\xrightarrow{}_i$ being the transition relation given by the semantics of T_i. Let $\bar{l}_0 = (l_1^0, \dots\ l_n^0)$ be the vector of initial locations. The semantics of the network is defined as a transition system $\langle S, s_0, \xrightarrow{}\rangle$, where $S = L_1 \times \dots \times L_n \times \mathbb{R}^C$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\xrightarrow{} \subseteq S \times \mathbb{R}^{\geq 0} \cup A \times S$ is the transition relation defined by:*

- $(\bar{l}, u) \xrightarrow{t} (\bar{l}, u+t)$, *if* $\forall t' : 0 \leq t' \leq t . u + t' \in I(\bar{l})$.
- $(\bar{l}, u) \xrightarrow{a_i} (\bar{l}[l'_i/l_i], u')$ *if an edge* $e_i = (l_i, a_i, g_i, r_i, l'_i) \in E_i$ *exists in* T_i *such that* $(l_i, u) \xrightarrow{a_i}_i (l'_i, u')$ *with* $u \in g_i$, $u' = [r_i \mapsto 0]u$ *and* $u' \in I(\bar{l}[l'_i/l_i])$.
- $(\bar{l}, u) \xrightarrow{c} (\bar{l}[l'_j/l_j, l'_i/l_i], u')$ *if two edges* $e_i = (l_i, c?, g_i, r_i, l'_i) \in E_i$ *and* $e_j = (l_j, c!, g_j, r_j, l'_j) \in E_j$ *exist such that* $u \in g_i \wedge g_j$, $u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(\bar{l}[l'_j/l_j, l'_i/l_i])$.

In the train gate example, a system with a train and a gate controller could follow the sequence of states $((\texttt{Train.Safe}, \texttt{Gate.Free}), 0) \xrightarrow{appr} ((\texttt{Train.Appr}, \texttt{Gate.Occ}), 0) \xrightarrow{15} ((\texttt{Train.Appr}, \texttt{Gate.Occ}), 15) \longrightarrow ((\texttt{Train.Cross}, \texttt{Gate.Occ}), 0) \dots$

4.2 Timed Automaton Templates

The implementation of the event-driven temporal logic is based on describing each formula as a network of timed automata that monitors the occurrence of events over time. As described in Section 3, formulae are evaluated against time bounded traces that execute in time intervals of the form $[t_i, t_f]$. In addition, formulae can include nested temporal operators whose evaluation can be restricted to a subinterval. For example, assume we want to evaluate $\langle \pi, t_i, t_f\rangle \vDash_e \Diamond_{[p,q]} \Box_{[r,s]} \phi$. The outer operator $\Diamond_{[p,q]}$ must find the different time intervals $[t_p, t_q] \subseteq [t_i, t_f]$, delimited by events p and q, to check if there is at least one satisfying the subformula $\Box_{[r,s]}\phi$. Similarly, given one of the time intervals $[t_p, t_q]$, the inner operator $\Box_{[r,s]}$ has to find all time intervals $[t_r, t_s] \subseteq [t_p, t_q]$ determined by events r and s to check whether ϕ holds in all of them.

We propose to construct a network of timed automata that includes a timed automaton to monitor each subformula. To this end, we have defined a set of timed automaton templates that describe the behaviour of each operator ($\vee, \mathcal{U}_{[p,q]}, \Diamond_{[p,q]}, \Box_{[p,q]}$, etc.). These templates can be instantiated to produce networks of timed automata representing complex formulae with nested operators.

The templates are parameterised with an `id` that identifies the template instance in the network of timed automata. The `id=0` is assigned to the complete formula, and the `id` is incremented for each subformula. Automaton templates synchronise using binary communication channels. Channels `start` and `stop` are used to synchronise each automaton with the interval t_i and t_f in which the (sub-)formula has to be evaluated. The channel `event` is used to synchronise events p and q. Since UPPAAL channels do not send data, we use variable `event_type` to know if the synchronisation event is p (`event_type==` 0) or q (`event_type==` 1), and the variable `event_timestamp` to record the time instant associated with the current event. Finally, channel `ready` is used to synchronise with the end the evaluation of a (sub-)formula. If the automaton `id` is greater than 0, it represents an operator nested inside a formula. In this case, channel `ready` is used to synchronise the automaton with the automaton of the outer operator. Otherwise, the automaton ends its execution in the `ok` or `error` location depending on the evaluation

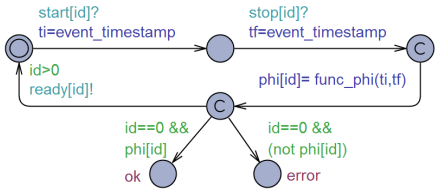


Figure 4: Timed automaton template of ϕ

```
bool func_phi(int ti, int tf)
{
    return
      ((ti!=-1)&& (tf!=-1)&&
       (tf - ti< 4));
}
```

Listing 1: Example of $\phi([t_i,t_j])$

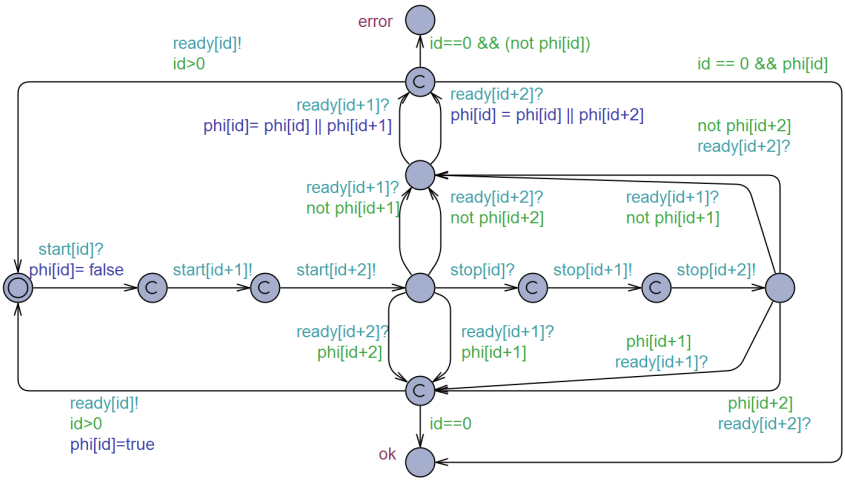


Figure 5: Timed automaton template of $\psi_1 \vee \psi_2$

result, which is stored in `phi`. In the following paragraphs, we present the templates describing each operator.

Interval formula (ϕ)

Figure 4 shows the automaton template of the *interval formula* ϕ. An interval formula is evaluated in the interval $[t_i,t_f]$ that is delimited by the synchronisation channels `start` and `stop`. After detecting the interval, the function $\phi([t_i,t_f])$ evaluates whether the interval formula holds and stores the result in `phi`. Listing 1 shows an example of function `func_phi(ti,tf)` that checks whether the interval length is less than 4 time units. We can modify the definition of this function as required by the specification.

Or ($\psi_1 \vee \psi_2$)

Figure 5 shows the template of the *or* operator. This automaton monitors if any of the two interval formulae (or sub-formulae) holds in the interval $[t_i,t_f]$. When the automaton synchronises with the `start` or `stop` channels, it instantaneously synchronises with the automata representing the sub-formulae, whose `id`s are `id+1` and `id+2`.

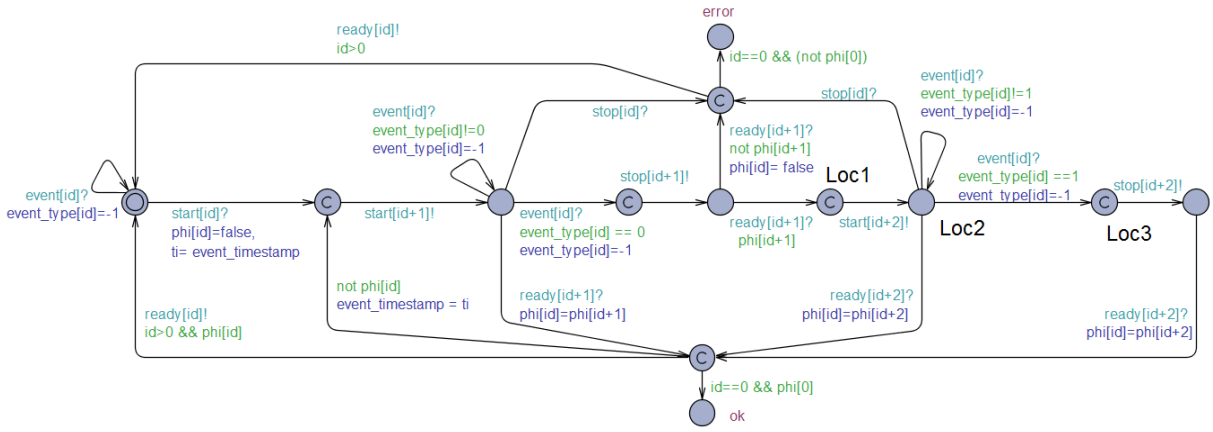


Figure 6: Timed automaton template of $\psi_1 \mathcal{U}_{[p,q]} \psi_2$

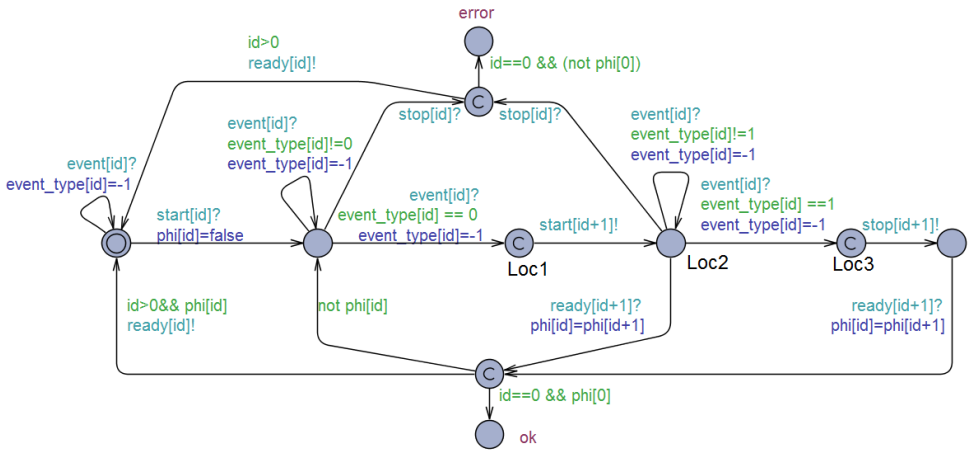


Figure 7: Timed automaton template of $\Diamond_{[p,q]} \psi$

Until ($\psi_1 \mathcal{U}_{[p,q]} \psi_2$)

Figure 6 shows the automaton template of the *until* operator. This automaton monitors the occurrence of event intervals $[p,q]$, and determines whether the interval formula (or sub-formula) holds from the beginning of the trace, that is, from t_i, until the second interval formula (or subformula) is satisfied according to the event interval $[p,q]$. The template of the operator $\psi_1 \mathcal{U}_{[p,-]} \psi_2$ is a modified version of this automaton, in which the location $Loc2$ is removed, as well as all its outgoing transitions, and the outgoing transition from $Loc1$ changes its destination to $Loc3$

Eventually ($\Diamond_{[p,q]} \psi$)

Figure 7 shows the template of the temporal operator *eventually* ($\Diamond_{[p,q]} \psi$). It monitors whether there exists at least one time interval $[t_p, t_q] \subseteq [t_i, t_f]$ satisfying $[p,q]$ on which the interval formula (or sub-formula) holds. Observe that this template is a simplification of the *until* template obtained by removing transitions dealing with ψ_1. Finally, the template of the operator $\Diamond_{[p,-]} \psi$ is built removing the location $Loc2$ and its outgoing transitions, and redirecting the outgoing transition from $Loc1$ to $Loc3$.

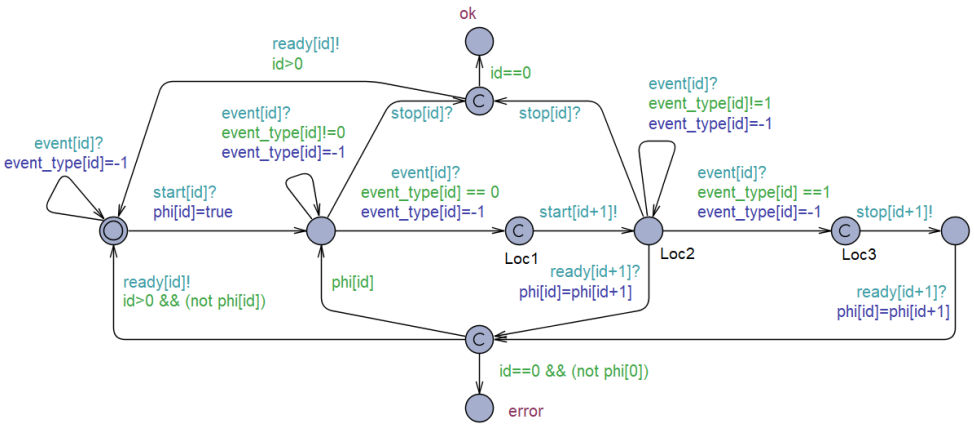


Figure 8: Timed automaton template of $\Box_{[p,q]}\psi$

Always ($\Box_{[p,q]}\psi$)

Figure 8 shows the template of the temporal operator *always*. It monitors whether all time intervals $[t_p,t_q] \in [t_i,t_f]$ determined by $[p,q]$ satisfy an interval formula (or sub-formula). This automaton is obtained from the *eventually* template. Since $\Box_{[p,q]}\psi \equiv \neg(\Diamond_{[p,q]}\neg\psi)$, the *always* template has the `ok` and `error` locations swapped, and the guards dealing with ψ negated. Finally, the template of the operator $\Box_{[p,-]}\psi$ is obtained in a similarly way to until and eventually automata.

4.3 Checking the Satisfaction of Formulae

Checking the satisfaction of a formula is reduced to a reachability problem. To determine if a trace, or a system model, satisfies a property described with the event-driven interval temporal logic, we analyse whether the associated network of timed automata reaches an specific location. In particular, when a formula is not satisfied, the automaton instance with `id=0` reaches the location named `error`, otherwise it reaches the location `ok`. Thus, the analysis consists of verifying with UPPAAL the following TCTL formula $A\,\Box\,\neg(Formula.error)$, where *Formula* is the automata instance with `id=0`.

The network of timed automata includes instances of the automata templates to represent the formula. The nesting relation of the operators is described with the template instance `id`. The automaton of the outermost operator has `id=0`. If a template instance with `id = x` has a nested sub-formula, the automaton describing the sub-formula will use `id = x+1`. In the case of *until* and *or* templates, which have two nested sub-formulae, the left sub-formula is identified with `id = x+1` and the right one with `id = x+2`. For instance, formula $\Box_{[p,q]}\,\Diamond_{[r,s]}\,\phi$ will be modelled with three automata. The automaton with `id=0` is an instance of the *always* operator, the instance of the *eventually* operator template has `id=1` and, finally, the instance of ϕ uses `id=2`. In the formula $\phi_1 \mathcal{U}_{[p,q]} \phi_2$, the automaton with `id=0` is an instance of the *until* operator, and `id=1` and `id=2` are, respectively, the instances of the first and second sub-formulas (ϕ_1 and ϕ_2).

The network of timed automata has to include a timed automaton that models the trace of events, or a more complex model describing a set of traces with the behaviour of the continuous variables monitored in the formula. This automaton sends the events through the synchronisation channels to the automata describing the formula. Figure 9 shows the model of the trace of events $s_0 \xrightarrow{\iota}_4 s_1 \xrightarrow{p}_d s_2 \xrightarrow{\iota}_5 s_3 \xrightarrow{p}_d s_4 \xrightarrow{\iota}_7 s_5 \xrightarrow{q}_d s_6 \xrightarrow{\iota}_{10} s_7 \xrightarrow{p}_d s_8 \xrightarrow{\iota}_{12} s_9 \xrightarrow{q}_d s_{10} \ldots$.

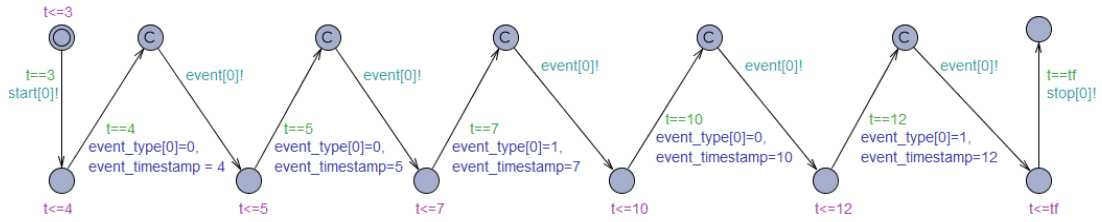


Figure 9: Model of an execution trace

4.4 Examples

In this section, we use the event-driven interval temporal logic to express some properties for the artificial pancreas case study [8]. These properties can be verified using the current implementation on UPPAAL.

Assume that we can measure the patient's blood sugar level every minute and be notified when the insulin pump injects a bolus (event `bolus`). In addition, assume that we can monitor when the patient goes to sleep (`sleep`) and wakes up (`wakeup`). The following formula describes the property: "*while the patient is sleeping, his/her sugar level is never below* 90 *mg/l*".

$$\Box_{[sleep,wakeup]}\,\phi, \qquad where\ \ \phi([t_i,t_f]) = \begin{cases} true & if\ \forall t \in [t_i,t_f],\ sugar(t) > 90mg/l \\ false & otherwise \end{cases} \tag{4.4}$$

This formula is equivalent to a network of timed automata with three instances of automaton templates. The automaton with `id=0` represents the operator $\Box_{[wakeup,sleep]}$ (see Figure 8), the automaton representing the atomic proposition (see Figure 4) has `id=1`, and finally, the execution trace automaton has `id=2`. Finally, to check whether a specific trace satisfies formula ((4.4)), we use the UPPAAL verifier to prove whether the automaton with `id=1` never reaches the `error` location.

We can express other interesting properties by nesting multiple operators. For instance, the following formula states that "*patient maintains a blood sugar level over 90mg/l while sleeping and each time the insulin pump injects a bolus the blood sugar level is greater than 140mg/l*".

$$\Box_{[sleep,wakeup]}\,(\phi_1 \wedge (\Box_{[bolus,-]}\phi_2)) \qquad where \qquad \phi_1([t_i,t_f]) = \begin{cases} true & if\ \forall t \in [t_i,t_f],\ sugar(t) \geq 90mg/l \\ false & otherwise \end{cases}$$

$$\phi_2([t_i,t_f]) = \begin{cases} true & if\ \forall t \in [t_i,t_f],\ sugar(t) \geq 140mg/l \\ false & otherwise \end{cases}$$

5 Conclusions

In this paper, we have presented an event-driven interval temporal logic suitable for describing properties in hybrid systems. The inspiration of the logic comes from the observation that many properties in continuous variables of hybrid systems can be more easily specified in terms of the time intervals where certain events take place. We have also constructed a prototype implementation of logic operators in such a way that each formula is translated into a network of timed automata that can be analysed on UPPAAL.

We have included a preliminary evaluation using a case study inspired by [8], which analyses the behaviour of an artificial pancreas.

This paper constitutes an initial approach to the construction of a testing architecture for hybrid systems involving model-based testing and runtime verification techniques.

Currently, we are interested in the application of the proposal to other domains. For instance, we have experience in the development of a similar approach that analyses extra-functional properties of mobile applications running in real devices. Although we do not have a model of the continuous behaviour of the mobile device, such as how energy is consumed, we can obtain this information from the TRIANGLE testbed [9], which monitors the evolution of different mobile phone magnitudes.

Finally, we plan to compare our proposal with other real-time logics in the literature, in particular in terms of expressiveness and complexity.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis & S. Yovine (1995): *The algorithmic analysis of hybrid systems*. *Theoretical Computer Science - Special issue on hybrid systems* 138(1), pp. 3–34.

[2] R. Alur & D. Dill (1991): *The Theory of Timed Automata*. In: *REX Workshop*, LNCS, Springer-Verlag, Berlin, Heidelberg, pp. 45–73.

[3] R. Alur, T. Feder & T. A. Henzinger (1996): *The Benefits of Relaxing Punctuality*. *J. ACM* 43(1), pp. 116–146.

[4] G. Behrmann, A. David & K. Larsen (2004): *A Tutorial on* UPPAAL. In: *Formal Methods for the Design of Real-Time Systems*, *LNCS* 3185, Springer–Verlag, pp. 200–237.

[5] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi & M. Hendriks (2006): *UPPAAL 4.0*. In: *3rd Int. Conference on the Quantitative Evaluation of Systems - (QEST'06)*, pp. 125–126, doi:10.1109/QEST.2006.59.

[6] J. A. Botia, A. Villa & J. Palma (2012): *Ambient Assisted Living system for in-home monitoring of healthy independent elders*. *Expert Systems with Applications* 39(9), pp. 8136 – 8148.

[7] M. Broy, B. Jonsson, J. P. Katoen, M. Leucker & A. Pretschner (2005): *Model-Based Testing of Reactive Systems: Advanced Lectures*. LNCS, Springer-Verlag New York, Inc.

[8] F. Cameron, G. Fainekos, D. M. Maahs & S. Sankaranarayanan (2015): *Towards a verified artificial pancreas: Challenges and solutions for runtime verification*, pp. 3–17. *LNCS* 9333, Springer Verlag.

[9] A. F. Cattoni, G. Corrales-Madueño, M. Dieudonne, P. Merino, A. Díaz-Zayas, A. Salmerón & et al. (2016): *An end-to-end testing ecosystem for 5G*. In: *European Conference on Networks and Communications, EuCNC 2016, Athens, Greece, June 27-30, 2016*, pp. 307–312, doi:10.1109/EuCNC.2016.7561053.

[10] Z. Chaochen & M. R. Hansen (2004): *Duration Calculus - A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. An EATCS Series, Springer.

[11] Thao Dang & Tarik Nahhal (2009): *Coverage-guided Test Generation for Continuous and Hybrid Systems*. *Form. Methods Syst. Des.* 34(2), pp. 183–213.

[12] G. Frehse (2008): *PHAVer: algorithmic verification of hybrid systems past HYTECH*. *Software Tools for Technology Transfer* 10(3), pp. 263–279.

[13] A. Goodloe, C.A. Muñoz, F. Kirchner & L. Correnson (2013): *Verification of Numerical Programs: From Real Numbers to Floating Point Numbers*. In: *5th Int. Symposium on NASA Formal Methods*, *LNCS* 7871, Springer, pp. 441–446.

[14] K. Havelund (2015): *Rule-based runtime verification revisited*. *International Journal on Software Tools for Technology Transfer* 17(2), pp. 143–170, doi:10.1007/s10009-014-0309-2.

[15] F. Lerda, J. Kapinski, H. Maka, E. M. Clarke & B. H. Krogh (2008): *Model checking in-the-loop: Finding counterexamples by systematic simulation*. In: *2008 American Control Conference*, pp. 2734–2740.

[16] M. Leucker (2012): *Teaching Runtime Verification*, pp. 34–48. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-29860-8_4.

[17] O. Maler & D. Ničković (2013): *Monitoring properties of analog and mixed-signal circuits*. *International Journal on Software Tools for Technology Transfer* 15(3), pp. 247–268.

[18] O. Maler, D. Nickovic & A. Pnueli (2005): *Real Time Temporal Logic: Past, Present, Future*. In: *Formal Modeling and Analysis of Timed Systems*, Springer Berlin Heidelberg.

[19] D. Ničković, O. Lebeltel, O. Maler, T. Ferrère & D. Ulus (2018): *AMT 2.0: Qualitative and Quantitative Trace Analysis with Extended Signal Temporal Logic*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, pp. 303–319.

[20] Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon & G. Kutty (1996): *Interval logics and their decision procedures: Part I: An interval logic*. *Theoretical Computer Science* 166(1), pp. 1 – 47.

[21] Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon & G. Kutty (1996): *Interval logics and their decision procedures: Part II: a real-time interval logic*. *Theoretical Computer Science* 170(1), pp. 1 – 46.

[22] R. L. Schwartz, P. M. Melliar-Smith & F. H. Vogt (1983): *An Interval Logic for Higher-level Temporal Reasoning*. In: *Procs. of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, ACM, New York, NY, USA, pp. 173–186.