6 Timed logics

The region automaton abstraction is sound for verifying reachability properties. This is due to the time-abstract bisimulation property which links the region automaton and the original timed automaton. As a consequence, the region automaton abstraction can be used to verify all properties that are invariant by time-abstract bisimulation. This is for instance the case of safety properties, of $\omega$-regular properties, or of untimed properties expressed in LTL [Pnu77] or in CTL [CE81]. However, this construction cannot be directly used to verify properties expressed in a timed temporal logic like TCTL [ACD90, ACD93] because a property like “reaching a state in exactly 5 units of time” is not invariant by time-abstract bisimulation. However we show that it can nevertheless be used.

6.1 Timed extensions of CTL

We let $\mathcal{AP}$ be a finite set of atomic propositions. In this section, timed automata are tuples $\mathcal{A} = (L, \ell_0, X, T, \mathcal{L})$ such that $(L, \ell_0, X, T, \mathcal{L})$ is a standard timed automaton with a single initial location, no final locations, no alphabet, and $\mathcal{L} : L \rightarrow 2^{\mathcal{AP}}$ is a labelling function.

Syntax and semantics. The branching-time logic TCTL,\(^\text{12}\) which extends the classical untimed branching-time logic CTL with time constraints on modalities, has been defined. The syntax of TCTL is given by the following grammar:

$$\text{TCTL} \ni \phi ::= a \mid \neg \phi \mid \phi \lor \psi \mid E \phi U_I \psi \mid A \phi U_I \psi$$

where $a \in \mathcal{AP}$, and $I$ is an interval of $\mathbb{R}_+$ with integral bounds.

There are two possible semantics for TCTL, one which is said ‘continuous’, and the other one which is more discrete and is said ‘pointwise’. These two semantics share rules for basic modalities, and only differ in the interpretation of the term ‘position’:

- $(\ell, v) \models a \iff a \in \mathcal{L}(\ell)$
- $(\ell, v) \models \neg \phi \iff (\ell, v) \not\models \phi$
- $(\ell, v) \models \phi \lor \psi \iff (\ell, v) \models \phi$ or $(\ell, v) \models \psi$
- $(\ell, v) \models E \phi U_I \psi \iff$ there is an infinite run $\rho$ in $\mathcal{A}$ from $(\ell, v)$ such that $\rho \models \phi U_I \psi$
- $(\ell, v) \models A \phi U_I \psi \iff$ any infinite run $\rho$ in $\mathcal{A}$ from $(\ell, v)$ is such that $\rho \models \phi U_I \psi$
- $\rho \models \phi U_I \psi \iff$ there exists a position $\pi > 0$ along $\rho$ such that $\rho[\pi] \models \psi$, for every position $0 < \pi' < \pi$, $\rho[\pi'] \models \phi$, and duration($\rho_{\leq \pi}$) $\in I$

where $\rho[\pi]$ is the state of $\rho$ at position $\pi$, $\rho_{\leq \pi}$ is the finite prefix of $\rho$ ending at position $\pi$, and duration($\rho_{\leq \pi}$) is the sum of all delays along $\rho$ up to position $\pi$. The $U$-modality is called the ‘Until’ operator.

\(^\text{12}\) TCTL stands for “Timed Computation Tree Logic” and has been defined in [ACD90].
In the continuous semantics, a position in a run $\rho$ is any state appearing along $\rho$. For instance if there is a transition $(\ell, v) \xrightarrow{\tau,e} (\ell', v')$ in $\rho$, then any state $(\ell, v + t)$ with $0 \leq t \leq \tau$ is a position of $\rho$, and obviously, so is $(\ell', v')$. This semantics is very strong because for $\rho$ to satisfy $\phi \mathcal{U} \sim c \psi$, all intermediary states of $\rho$ need to satisfy $\phi$ before $\psi$ holds.

In the pointwise semantics, a position in a run $\rho = s_0 \xrightarrow{\tau_1,e_1} s_1 \xrightarrow{\tau_2,e_2} s_2 \cdots s_{n-1} \xrightarrow{\tau_n,e_n} s_n \cdots$ is an integer $i$ and the corresponding state $s_i$. In this semantics, formulas are checked only right after a discrete action has been done. Sometimes, the pointwise semantics is given in terms of actions and timed words, but it does not change anything. Later, we may sometimes use the timed words terminology.

As usually in CTL, we define syntactic sugar to TCTL: $\top \equiv a \lor \neg a$ standing for true, $\bot \equiv \neg \top$ standing for false, the implication $\varphi \rightarrow \psi \equiv (\neg \varphi \lor \psi)$, the eventuality operator $F_I \phi \equiv \top \mathcal{U} I \phi$, and the globally operator $G_I \phi \equiv \neg (F_I \neg \phi)$.

**Example 8.** In TCTL, we can write many kinds of properties, for instance, bounded-response time properties like

$$\mathsf{AG}(a \rightarrow \mathsf{AF}_{\leq 56} b)$$

expressing that each time $a$ holds, along all possible runs, $b$ has to hold within 56 time units.

**Remark 9.** In [HNSY94], TCTL is given with external clock variables. That is, we can use variables to express timing constraints. We will not give the precise grammar of that version of TCTL, but just give the equivalent of formula (1) in this framework:

$$\mathsf{AG} (a \rightarrow x.\mathsf{AF} (b \land x \leq 56))$$

The interpretation of that formula is the following: each time an $a$ is encountered, we reset a clock $x$, and check that along all possible runs, later, $b$ holds and the value of the clock $x$ (which has increased at the same speed as the universal time) is not more than 56.

In [BCM05], it has been proved that TCTL with external clock variables is strictly more expressive than TCTL with intervals constraining the modalities.

**Decidability.** The model-checking problem asks, given a timed automaton $\mathcal{A}$ and a TCTL formula $\phi$, whether $\mathcal{A}$ satisfies $\phi$ from its initial configuration $(\ell_0, 0_x)$.

**Theorem 3.** For the two semantics, the model-checking problem for TCTL is PSPACE-complete.

**Proof.** The hardness comes from the complexity of deciding reachability properties in timed automata (which can be expressed using formula $\mathsf{EF} q_f$, where $q_f$ is an atomic proposition labelling the final state).

For the upper bound, we fix a diagonal-free timed automaton $\mathcal{A} = (L, \ell_0, X,T, \mathcal{L})$, and we consider its corresponding region equivalence, which we denote $\equiv$. We do the proof in the framework of the pointwise semantics, but it can be extended to the continuous semantics. We show the following lemma:

\[\text{That can be formalized, see for instance [BBBL05].}\]
Lemma 8. Let $\phi$ be a TCTL formula. For every $\ell \in L$, for all $v, v' \in \mathbb{T}^X$ such that $v \equiv v'$,

$$
A, (\ell, v) \models \phi \iff A, (\ell, v') \models \phi
$$

Proof. We do the proof by a structural induction on $\phi$. We therefore assume that $\phi = E \phi_1 U I \phi_2$, where the $\phi_i$'s are equally satisfied within a region.

We fix a constant $M$, which is larger than the finite bounds defining $I$. We define a new set of clocks $\bar{X} = X \cup \{u\}$, where $u$ is a fresh clock. This clock will be used to measure the duration of the execution, which needs to lie in $I$.

We then consider the region equivalence over $\bar{X}$, denoted $\bar{\equiv}$, which refines $\equiv$ by adding clock $u$ with maximal constant $M$.

We now assume that $A, (\ell, v) \models E \phi_1 U I \phi_2$. This means that there is an execution in $A$

$$
\rho = (\ell, v_0) \xrightarrow{d_1, e_1} (\ell_1, v_1) \xrightarrow{d_2, e_2} \ldots \xrightarrow{d_k, e_k} (\ell_k, v_k) \rightarrow \ldots \text{such that } v_0 = v \text{ and } \rho \models \phi_1 U I \phi_2.
$$

Expanding $v_i$ into $\bar{v}_i$ over $\bar{X}$ such that $\bar{v}_i(x) = v_i(x)$ for every $x \in X$ and $\bar{v}_i(u) = \sum_{i=1}^{j \leq d_j} d_{j \leq d_j}$, we get an execution $\bar{\rho} = (\ell, v_0) \xrightarrow{d_1, e_1} (\ell_1, \bar{v}_1) \xrightarrow{d_2, e_2} \ldots \xrightarrow{d_k, e_k} (\ell_k, \bar{v}_k) \rightarrow \ldots \text{such that } \bar{\rho} \models \phi_1 U I \phi_2 \text{ (since clock } u \text{ is not constrained in } A)$. There exists $k$ such that $(\ell_k, v_k) \models \phi_2$, for every $1 \leq j < k$, $(\ell_j, v_j) \models \phi_1$, and $\bar{v}_k(u) \in I$ ($\bar{v}_k(u)$ is the duration of the run $\rho$ up to position $k$).

Pick $v'$ such that $v \equiv v'$, write $v'_0 = v'$, and define $\bar{v}'_0$ such that $\bar{v}'_0(x) = v'_0(x)$ if $x \in X$ and $\bar{v}'_0(u) = 0$. Then $\bar{v}_0 \bar{\equiv} \bar{v}'_0$. Hence one can build an execution $\bar{\rho}' = (\ell, v'_0) \xrightarrow{d_1, e_1} (\ell_1, \bar{v}'_1) \xrightarrow{d_2, e_2} \ldots \xrightarrow{d_k, e_k} (\ell_k, \bar{v}'_k) \rightarrow \ldots \text{such that for every } i \geq 0, \bar{v}'_i \bar{\equiv} \bar{v}_i$. Define $v'_i$ the projection of $\bar{v}'_i$ over $X$. Then, $v_i \equiv v'_i$. It holds that $(\ell_k, v'_k) \models \phi_2$, and for every $1 \leq j < k$, $(\ell_j, v'_j) \models \phi_1$. Now, since $\bar{v}_k \equiv \bar{v}'_k$, we also have that $\bar{v}_k(u) \in I$, which implies that $\rho'$ defined as the projection of $\bar{\rho}'$ over $X$, satisfies $\phi_1 U I \phi_2$. This implies the expected result that $A, (\ell, v') \models E \phi_1 U I \phi_2$.

Using a labelling algorithm as for CTL, we can therefore decide the model-checking problem for TCTL. By guessing and recomputing the truth for subformulas on-demand, we can improve the algorithm and make it use polynomial space only (witness paths have at most exponential length). This concludes the proof.

6.2 Timed extensions of LTL

Like classical temporal logics, linear-time property languages have also been studied in the framework of timed systems. Two main temporal formalisms have been defined: (i) MTL14 is the counterpart of TCTL without external clock variables, and extends LTL [Pnu77] by adding timing constraints on modalities; (ii) TPTL15 extends LTL by adding timing constraints to specifications using external variables and constraints thereon.

\[^{14}\text{MTL} \text{ stands for "Metric Temporal Logic" and has been first proposed by Koymans [Koy90].}\]

\[^{15}\text{TPTL} \text{ stands for "Timed Propositional Temporal Logic", and has been first proposed by Alur and Henzinger [AH89,AH94].}\]
Syntax and semantics of MTL. The syntax of MTL is given by the following grammar:

\[ MTL \ni \phi :::= a \mid \neg \phi \mid \phi \lor \psi \mid \phi \lor I \psi \]

where \( a \in \text{AP} \), and \( I \) is an interval of \( \mathbb{R}_+ \) with integral bounds.

It will even be more important in the context of linear-time timed temporal logics: as for TCTL, we distinguish between the two semantics, pointwise and continuous. Let \( s_0 \xrightarrow{\tau_1, e_1} s_1 \xrightarrow{\tau_2, e_2} s_2 \cdots s_{n-1} \xrightarrow{\tau_n, e_n} s_n \cdots \) with \( s_0 = (\ell_0, v_0) \) be a finite or infinite run. Then:

\[ \rho \models a \iff a \in L(\ell_0) \]
\[ \rho \models \neg \phi \iff \rho \not\models \phi \]
\[ \rho \models \phi \lor \psi \iff \rho \models \phi \text{ or } \rho \models \psi \]
\[ \rho \models \phi \lor I \psi \iff \text{there exists a position } \pi > 0 \text{ along } \varrho \text{ s.t.} \]
\[ \varrho[\pi] \models \psi, \text{ for every position } 0 < \pi' < \pi, \]
\[ \varrho[\pi'] \models \phi, \text{ and duration}(\varrho[\pi]) \in I \]

with the same distinctions for the term ‘position’, depending on the choice of the semantics.

As for LTL, we define some syntactic sugar for MTL: \( \mathbf{t} \equiv (a \lor \neg a) \) stands for true, \( \mathbf{f} \equiv (\neg \mathbf{t}) \) stands for false, \( (\varphi \rightarrow \psi) \equiv (\neg \varphi \lor \psi) \), \( F_I \varphi \equiv (\mathbf{t} \lor I \varphi) \) (eventually, \( \varphi \) will hold within interval \( I \) from now), \( G_I \varphi \equiv \neg (F_I \neg \varphi) \) (for all positions within \( I \), \( \varphi \) holds), and \( X_I \varphi \equiv (\mathbf{f} \lor I \varphi) \) (next position is within \( I \) from now and satisfies \( \varphi \)). Moreover, we use pseudo-arithmetical expressions to represent intervals. For instance, ‘\( = 1 \)’ stands for the singleton interval \([1, 1]\), and ‘\( \geq 2 \)’ stands for the interval \([2, +\infty)\).

Example 9. Using MTL, we can write properties like

\[ G (\text{problem} \rightarrow F_{\leq 56} \text{alarm}) \]

expressing that each time a problem occurs, within 56 time units, an alarm rings.

We can also express more involved properties, like

\[ G (\text{problem} \rightarrow (F_{\leq 15} \text{repair} \lor F_{[12, 15]} \text{alarm})) \]

which expresses that each time a problem occurs, then either it is repaired in no more than 15 time units, or an alarm rings for 3 time units 12 time units after the problem. There is no direct and obvious way to express this kind of property in TCTL.

Remark 10. The choice of the interpretation of MTL in terms of the pointwise or of the continuous semantics has a large impact on the meaning of the formulas, and as we will see later, also on their applicability in model-checking. The formula \( F_{=2} a \) expresses that an \( a \) will happen two time units later. In the continuous semantics, this formula is equivalent to \( F_{=1} F_{=1} a \) (in one time unit, it will be the case that in one time unit, an \( a \) occurs). However, it is not the case in the pointwise semantics, as there may be no action one time unit later, hence any formula \( F_{=1} \psi \) would be evaluated as wrong from the initial point.
Two extensions: TPTL and MTL+Past

In the following, we will also consider two extensions of MTL. First, as for TCTL, external clock variables can be used to express timing constraints. For instance, property (2) can be written as

\[ G(\text{problem} \to x. F(\text{alarm} \land x \leq 56)) \]

where \( x \) is a fresh variable which is reset when a problem occurs, and whose value is checked to be within \([0, 56]\) when the alarm rings. The value of \( x \) is supposed to evolve at the same speed as the universal time (similar to a clock in a timed automaton). This logic with external clock variables is called TPTL, and has been first proposed in [AH89]. We give another example of formulas that can be expressed in TPTL:

\[ G(\text{problem} \to x. F(\text{alarm} \land F(\text{failsafe} \land x \leq 56))) \] (3)

This formula says that whenever a problem occurs, then within 56 time units, an alarm rings and later (but still within 56 time units since the problem occurred), the system enters a failsafe mode. It has been proved in [BCM05] that TPTL is strictly more expressive than MTL. For the pointwise semantics, formula (3) is a witness to that expressiveness result, meaning that formula (3) cannot be expressed in MTL. Surprisingly, this formula has an equivalent formula in MTL,\(^{16}\) and a more involved formula has been proposed to distinguish between MTL and TPTL.

Following the classical untimed framework [Kam68,LPZ85], we also extend MTL with past-time modalities, i.e., with the ‘Since’ modality, somewhat the dual of the ‘Until’ modality: formula \( \varphi S_I \psi \) expresses that \( \varphi \) holds since \( \psi \) was true (within \( I \) in the past). In the following, we will only use the simple formula \( F_I^{-1} \varphi \) which is the dual of \( F_I \varphi \) for the past: it expresses that \( \varphi \) was true in the past, within a delay belonging to the interval \( I \). For instance, the formula

\[ G(a \to F_{-1}^{-1} b) \]

expresses that every \( a \) is preceded one time unit earlier by a \( b \). This logic is denoted MTL+Past.

6.3 The model-checking problem

The model-checking problem asks, given \( A \) a timed automaton and \( \varphi \) a formula, whether \( A \) satisfies \( \varphi \), written \( A \models \varphi \), and meaning that all (accepting) runs of \( A \) satisfy the formula \( \varphi \). We then give hints for understanding the following (un)decidability results:

Theorem 4. Over finite runs, the model-checking problem for:

\[ F_{\leq 28} \text{alarm} \land F_{[28,56]} \text{failsafe} \]
\[ \lor F_{\leq 28} (\text{alarm} \land F_{\leq 28} \text{failsafe}) \]
\[ \lor F_{\leq 28} (F_{\leq 28} \text{alarm} \land F_{<28} \text{failsafe}) \]

\(^{16}\) Indeed, we can prove (cf [BCM05]) that formula \( x. F(\text{alarm} \land F(\text{failsafe} \land x \leq 56)) \) is equivalent to
From these (un)decidability results, we learn that model-checking linear-time timed temporal logic is hard! And much harder than branching-time timed temporal logic. This is already the case in the untimed framework, but the gap dramatically increases in the timed framework. We first explain why it is so hard to model-check linear-time timed temporal logics. For that, we follow ideas developed in [Che07].

Model-checking linear-time timed properties is hard

We first explain the non-primitive recursive lower bound for the MTL model-checking problem, which relies on the halting problem for channel machines with insertion errors. A channel machine is a finite automaton which can write on a channel and read from it following a FIFO policy. We note ‘a!’ for writing a at the tail of the channel and ‘a?’ for reading an a at the head of the channel. A channel machine has insertion errors if any letter can be written at any time anywhere in the channel. A channel machine without insertion errors is said perfect, or insertion-free.

Example 10. Consider the channel machine depicted on the next figure:

A configuration of this system is a pair (s, w) where s is a discrete state of the machine and w is a word representing the content of the channel. We give an error-free computation example for that machine:

\[(s_1, \varepsilon) \xrightarrow{a!} (s_1, a) \xrightarrow{a!} (s_1, aa) \xrightarrow{b?} (s_2, ab) \xrightarrow{a?} (s_3, ab) \xrightarrow{a?} (s_3, b) \xrightarrow{b?} (s_4, \varepsilon)\]

We can see that no error-free computation allows to reach state s_5 (because no c is written on the channel). If we assume that this machine has insertion errors, then the following move is allowed:

\[(s_4, \varepsilon) \xrightarrow{c?} (s_5, \varepsilon)\]

(we assume implicitly that a c has been inserted on the channel, so that the last transition labelled by ‘c?’ can now be fired).

Given a channel machine C with a distinguished final state, the halting problem asks whether the machine C has an execution halting in the final state. The hardness results stated in Theorem 4 will be proved by reduction to the following problems about channel machines.
Proposition 6. – The halting problem is undecidable for channel systems [BZ83].
– The halting problem is non-primitive recursive for channel machines with insertion errors [Sch02].

We now explain how MTL (and variants) can capture the behaviours of channel machines. The idea is to encode a computation of a channel machine as a timed word. In this encoding, the underlying untimed word is the trace of the computation, that is, an alternating sequence of states and actions. We use timing constraints to enforce the channel be FIFO: we require that any write action ‘a!’ is followed one time unit later by a corresponding read action ‘a?’.

The above timed word encodes the following computation of the channel machine:

\[(q_0, \varepsilon) \xrightarrow{a!} (q_1, a) \xrightarrow{b!} (q_2, ab) \xrightarrow{a?} (q_3, b) \xrightarrow{c!} (q_4, bc) \xrightarrow{b?} (q_5, c) \cdots\]

To properly encode a behaviour of a channel machine, a timed word must satisfy the following constraints:

– states and actions alternate. This can be checked using an LTL formula.

– the untimed projection of the timed words follows the rules of the channel machine. This can also be encoded with an LTL formula.

– the channel is FIFO: to do that we express that every write action ‘a!’ is followed one time unit later by a corresponding read action. This can be expressed in MTL using formulas of the form:

\[G (a! \rightarrow F_{=1}a?)\]

However, this formula does not encode the property that the channel behaves properly. Indeed, nothing prevents a read event ‘a?’ to happen, even though there is no corresponding write event ‘a!’ one time unit earlier. For instance, consider the following timed word:

\[\cdots\]

Formally, in [Sch02], that’s the halting problem for lossy channel machines which is proved non-primitive recursive, but this is indeed equivalent.
This timed word satisfies the propagating formulas $G (a! \rightarrow F_{=1}a?)$ (for every letter $a$), even though the event ‘c?’ is not preceded by any action one unit earlier. The above formula hence only encodes the behaviour of a channel machine with insertion errors. However, from that study, we already learn that the model-checking of MTL (over finite words) is non-primitive recursive. To encode a perfect channel machine, we need to be able to express the property that every ‘a?’ is preceded one time unit earlier by an ‘a!’.

We call this property the ‘backward matching property’.

We now discuss how we can express the backward matching property in timed temporal logics. Indeed, we would like to know whether MTL can express or not the behaviour of a perfect channel machine. We will present here natural ideas, which will happen to be wrong for MTL, but sufficient to prove undecidability of several variants or extensions of MTL.

- A first simple idea is to express this ‘backward matching property’ using the following formula:

$$G ((F_{=1}a?) \rightarrow a!)$$

which expresses the fact that if there is a read event ‘a?’ one time unit later, then there must be right now a corresponding write event ‘a!’.

It is not hard to see that in the pointwise semantics, this does not express what we want. Indeed this formula is still satisfied by the above-mentioned timed word, because there is no action one time unit before the action ‘c?’.

However, in the continuous semantics, this formula really enforces a perfect behaviour of the FIFO channel. That is why MTL in the continuous semantics has an undecidable model-checking problem.

- A second idea is to express this ‘backward matching property’ using a past-time modality (hence in MTL+Past). The formula

$$G (a? \rightarrow F_{=1}a!)$$

precisely expresses that every read event ‘a?’ is preceded one time unit earlier by a matching write event ‘a!’.

That is why MTL+Past is undecidable, even in the pointwise semantics.

- Finally, the ‘backward matching property’ can be expressed in TPTL using the following more involved property:

$$\neg \left( Fx \cdot X (y \cdot F(x > 1 \land y < 1 \land a?)) \right)$$

Informally, this formula negates the fact that there are two consecutive positions (in the pointwise sense) such that an $a$ is read more than one time unit after the first position, and less than time unit after the second position. This precisely negates the fact that there is an ‘a?’ not preceded one time unit earlier by an action. This implies that TPTL is undecidable, already in the pointwise semantics (when at least two clock variables are used).
From all these considerations, we get that in the pointwise semantics, over finite runs, we can only express channel machines with insertion errors with MTL, whereas perfect channel machines can be expressed either using MTL in the continuous semantics, or using MTL+Past or TPTL in the pointwise semantics (both over finite words). This concludes the hardness results stated in Theorem 4.

**MTL model-checking over finite words is decidable** We now explain how we can prove the decidability of MTL over finite words in the pointwise semantics. We know that LTL formulas can be transformed into alternating finite automata [MSS88,Var96]. In a similar way, we can transform any formula of MTL into an alternating timed automaton [LW05] with a single clock [OW05]. For instance, the formula $G_{<2}(a \rightarrow F_{=1}b)$ can be transformed into the following alternating timed automaton:

```
 r -> x<2; a x=0
 a -> x=1; b
```

with the obvious interpretation that any time an $a$ is done (within the two first time units), we fork a new thread which will check that a $b$ appears one time unit later. A behaviour of this alternating timed automaton is an unbounded tree, and it is not obvious that it is possible to check for emptiness of such a system. Indeed, checking emptiness of alternating timed automata is decidable only for one clock over finite timed words, any slight extension (infinite timed words, two clocks, silent moves\textsuperscript{18}) leading to undecidability [LW08,OW07].

We explain how we can however understand the decidability of this model [OW05]. Consider the timed word $(c, 0.6)(a, 0.7)(a, 1.5)(b, 1.7)$. The execution of the above alternating timed automaton on that timed word can be depicted as the following tree, which is not accepting as one of the branches (the second one on the picture) is not accepting (accepting states are underlined).

```
 r, 0 -> c r, 0.6 a r, 0.7 a r, 1.5 b r, 1.7
 s, 0 s, 0 s, 0.2 s, 0.8 z
```

A configuration of the alternating timed automaton is a slice of the tree, for instance, $\{(r, 1.5), (s, 0), (s, 0.8)\}$ is a configuration. Because we consider finite words, there is no need to consider the tree structure of the execution, but we can reason globally on configurations of the automaton. There are infinitely many such configurations, but as for the region automaton construction for timed automata [AD94], the precise values of the clocks is not really relevant, and the things which are important in a configuration are the integral parts of the clocks and the relative order of the fractional parts of the clocks. For instance, for

\textsuperscript{18} Or $\varepsilon$-transitions, if we follow the classical terminology in formal language theory.
the above-mentioned configuration, we only need to know that there is a state \((s, 0)\) with fractional part 0, and two other states \((s, 0)\) and \((r, 1)\) such that the fractional part for \((s, 0)\) is greater than the fractional part for \((r, 1)\). For all configurations with the same abstraction, the possible future behaviours are the same, in a time-abstract bisimulation sense [OW07,LW08]. Unfortunately, the set of abstractions of possible configurations of the alternating timed automaton is also infinite. The most important property is then that there is a well-quasi-order on the set of abstractions of configurations, and that we can use it to provide an algorithm to decide emptiness [FS01]. This briefly sketches an algorithm for deciding the MTL model-checking problem over finite timed words (in the pointwise semantics).

Note that we can prove the decidability of TPTL with a single internal variable applying the same method.
7 The language-theoretic perspective

In the previous section we have presented the region automaton abstraction, which can be used to model-check several kinds of simple properties, like reachability properties. From a language perspective, this means that the emptiness problem is decidable for timed automata. In this section we study further language-theoretic properties of timed languages accepted by timed automata, and show in particular some negative results.

7.1 Boolean operations

Closure under Boolean operations is a basic property which is interesting for modelling and verification reasons.

Proposition 7. The class of timed languages accepted by timed automata is closed under finite union and finite intersection.

Proof (Sketch of proof). Closure under finite union is rather straightforward by taking the disjoint union of all timed automata.

The closure under finite intersection follows the lines of the standard product construction used in the case of finite automata. Only clock constraints, invariants and resets of clocks need be carefully handled. We illustrate the general construction with the intersection of two timed automata $A_1 = (L_1, L_{10}, X_1, \Sigma, T_1, \operatorname{Inv}_1)$ and $A_2 = (L_2, L_{20}, X_2, \Sigma, T_2, \operatorname{Inv}_2)$ over a single alphabet $\Sigma$. We assume that the two sets of clocks $X_1$ and $X_2$ are disjoint (otherwise we rename clocks so that it is actually the case). Then we define the timed automaton $A = (L, L_0, X, \Sigma, T, \operatorname{Inv})$ by:

- $L = L_1 \times L_2$, $L_0 = L_{10} \times L_{20}$, $L_F = L_{1F} \times L_{2F}$;
- $X = X_1 \cup X_2$ (disjoint union);
- the set $T$ is composed of transitions of the form $(\ell_1, \ell_2) \xrightarrow{g,a,Y} (\ell'_1, \ell'_2)$ whenever there exist two transitions $\ell_1 \xrightarrow{g_1,a,Y_1} \ell'_1$ in $T_1$ and $\ell_2 \xrightarrow{g_2,a,Y_2} \ell'_2$ in $T_2$ such that:
  - $g = g_1 \land g_2$;
  - $Y = Y_1 \cup Y_2$;
- $\operatorname{Inv}((\ell_1, \ell_2)) = \operatorname{Inv}_1(\ell_1) \land \operatorname{Inv}_2(\ell_2)$.

This is straightforward to prove that a timed word is accepted by $A$ iff it is both accepted by $A_1$ and $A_2$.

The following proposition is on the contrary rather bad news.

Proposition 8. The class of timed languages accepted by timed automata is not closed under complementation.

The most well-known timed automaton, already given in [AD94], which cannot be complemented is given in Figure 6. This automaton, over the single-letter alphabet $\{a\}$, recognizes the timed language:

$$\{(a, t_1)(a, t_2) \ldots (a, t_n) \mid n \in \mathbb{N}, n \geq 2 \text{ and there exist } 1 \leq i < j \leq n \text{ with } t_j - t_i = 1\}$$
Intuitively, to be recognized by a timed automaton, the complement of this timed language would require an unbounded number of clocks, because for any action $a$, we need to check that there is no $a$-action one time unit later, so a fresh clock is intuitively required. However the complete proof is rather technical and harassing [Bou98], and we do not provide it here.

An alternative and elegant proof of the above proposition has been proposed in [AM04], and this is the one we have decided to present here.

**Proof.** We consider the timed automaton of Figure 7. It accepts the following timed language over the alphabet \{a, b\}:

$$L = \{(\alpha_1, t_1) \ldots (\alpha_n, t_n) \mid n \in \mathbb{N}, n \geq 1, \exists 1 \leq i \leq n \text{ s.t. } \alpha_i = a \text{ and } \forall i < j \leq n, t_j - t_i \neq 1\}$$

We assume towards a contradiction that $\overline{L}$ (the complement of $L$) can be recognized by a timed automaton. It is not hard to get convinced that the timed language over the alphabet \{a, b\}

$$L' = \{(a^+b^*, \tau) \mid \text{all } a's \text{ happen before } 1 \text{ and no two } a's \text{ simultaneously}\}$$

is accepted by the timed automaton:

$$y > 0, x < 1, a, y := 0 \quad \quad b$$

Hence by Proposition 7, $\overline{L} \cap L'$ is accepted by some timed automaton. The following lemma is just a matter of expanding and manipulating the definition of $\overline{L} \cap L'$.

**Lemma 9.** The untiming of $\overline{L} \cap L'$ is the non-regular language

$$\{a^nb^m \mid \text{n, m } \geq 1\}.$$
This lemma yields a contradiction with the fact that $L \cap L'$ is accepted by some timed automaton, say $\mathcal{B}$, because the untiming of $L \cap L'$ should then be recognized by the region automaton of $\mathcal{B}$. Hence we conclude that the complement of $L$ is not recognized by any timed automaton.

7.2 The universality and inclusion problems

The universality problem asks, given a timed automaton $\mathcal{A}$, whether $\mathcal{A}$ accepts all (finite) timed words. The inclusion problem asks, given two timed automata $\mathcal{A}$ and $\mathcal{B}$, whether all timed words accepted by $\mathcal{B}$ are also accepted by $\mathcal{A}$, that is whether $L(\mathcal{B}) \subseteq L(\mathcal{A})$. Note that the universality problem is a special instance of the inclusion problem, where $\mathcal{B}$ is universal, i.e. accepts all (finite) timed words. The following result is bad news in the verification context, as argued in Section 3.

**Theorem 5 ([AD90,AD94]).** The universality problem is undecidable for timed automata.

**Proof.** We encode the halting problem for perfect channel machines as a universality problem of a timed automaton.

Let $\mathcal{M}$ be a channel machine. We use the same encoding as for the logics MTL, that is, a (finite) feasible execution\(^{19}\)

$$(q_0, \epsilon) \xrightarrow{\alpha_1} (q_1, w_1) \ldots \xrightarrow{\alpha_n} (q_n, w_n)$$

of $\mathcal{M}$ will be encoded by the (finite) timed word

$$(q_0, t_0)(\alpha_1, t_1)(q_1, t_2) \ldots (\alpha_n, t_{2n-1})(q_n, t_{2n})$$

such that:

1. $(t_i)_{0 \leq i \leq 2n}$ is (strictly) increasing;
2. for every $i$ such that $\alpha_i = a!$, there exists $j > i$ such that $\alpha_j = a?$ and $t_j = t_i + 1$
3. for every $j$ such that $\alpha_j = a?$, there exists $i < j$ such that $\alpha_i = a!$ and $t_i = t_j - 1$

We build a timed automaton $\mathcal{A}$ (over finite words) which will accept all (finite) timed words which are not encodings of a halting computation of $\mathcal{M}$. The alphabet of $\mathcal{A}$ is:

$$Q \cup \{a?, a! \mid a \in \Sigma\}$$

where $Q$ is the set of states of $\mathcal{M}$ and $\Sigma$ is the alphabet of $\mathcal{M}$. We write $\tilde{\Sigma}$ for the set $\{a?, a! \mid a \in \Sigma\}$, $\Sigma? = \{a? \mid a \in \Sigma\}$ and $\Sigma! = \{a! \mid a \in \Sigma\}$.

This will be done using a highly non-deterministic timed automaton, which will deny one-by-one the various conditions:

- deny “correct alternations between states and actions, and follow the rules of $\mathcal{M}$”: complement the automaton of the machine $\mathcal{M}$.

\(^{19}\) If $\alpha_i = a?$, then the first letter of $w_i−1$ is $a$, and $w_i−1 = aw_i$; if $\alpha_i = a!$, then $w_i = w_i−1a$.
– deny “the sequence of timestamps is increasing”

\[ x := 0 \rightarrow x = 0 \]

– deny the timing constraints implementing the fifo rules

\[ x < 1 \]

\[ a! \]

\[ x := 0 \]

\[ \neq a? \]

\[ x = 1 \]

\[ x > 1 \]

\[ a? \]

\[ x := 0 \]

\[ y := 0 \]

\[ x > 1 \land y < 1 \]

We can show that \( \mathcal{A} \) is universal if and only if \( \mathcal{M} \) does not halt. Indeed, assume \( \mathcal{M} \) halts, and consider an accepting execution, and one of its corresponding encodings as a timed word. It satisfies all the rules, hence it cannot be accepted by \( \mathcal{A} \). Conversely pick a timed word which is not accepted by \( \mathcal{A} \). Then this word is a proper encoding of an halting execution in \( \mathcal{M} \), otherwise one of the conditions would be denied and the word would be accepted by \( \mathcal{A} \) ⊓ ⊔

The following is a straightforward corollary of the initial observation that the universality problem is a special instance of the inclusion problem.

**Corollary 1.** The inclusion problem is undecidable for timed automata.

It is interesting to notice that the reduction used in the above proof builds a timed automaton with two clocks. And actually, the universality problem (and also the inclusion problem) is decidable (but non-primitive recursive) for single-clock timed automata, see [ADOW05]. Recent developments have considered alternating timed automata (a natural extension of timed automata with alternations) [LW05,OW05,LW08,OW07], but Theorem 5 implies that the emptiness problem is undecidable for alternating timed automata.

### 7.3 Timed automata and determinism

In the context of formal languages, determinism is a standard and central notion which expresses that for a word there is at most one execution which reads that word. For regular languages determinism does not restrict recognition of languages, but for context-free languages this is not the case [HMU01]. We discuss in this section the issue of determinism in the context of timed automata, which gives some explanation to the previous negative results.
The class of deterministic timed automata. We give a syntactical definition of determinism in timed automata (with no invariants, for simplicity). A timed automaton $A = (L, L_0, L_F, X, \Sigma, T)$ is deterministic whenever $L_0$ is a singleton, and for every $\ell \in L$, for every $a \in \Sigma$, $(\ell, g_1, a, Y_1, \ell_1) \in T$ and $(\ell, g_2, a, Y_2, \ell_2) \in T$ imply $[g_1 \land g_2]_X = \emptyset$. This notion extends in a natural way the standard notion of determinism in finite automata.

In a deterministic timed automaton, for every timed word, there is at most one run that reads that timed word from a given configuration.

Example 11. The timed automaton of Figure 1 (see page 8) is deterministic. From location ‘alarm’, there are two outgoing transitions, but the constraints labelling those two transitions are disjoint. From the other locations, there is only one outgoing transition.

On the other hand, the timed automata of Figures 6 and 7 are not deterministic. In the first one, there is a non-deterministic choice from location $\ell_1$, but it can be removed by strengthening the constraint on the self-loop (adding one self-loop with the constraint $x < 1$ and another one with the constraint $x > 1$). There is another non-deterministic choice from location $\ell_0$, and this one cannot be removed (note that this is in general not obvious to see whether a non-deterministic choice can be removed or not!): it is not possible to predict when will be the occurrence of an $a$ that will be followed one time unit later by another $a$.

Deterministic timed automata form a strict subclass of timed automata.\textsuperscript{20} Using a product construction, as done in the proof of Proposition 7 for the intersection, it is easy to get convinced that this subclass is closed under finite union and finite intersection. On the other hand the two timed automata we have given to illustrate the non-closure under complementation of the class of standard timed automata (Proposition 8) are not deterministic. And actually it is not very hard to get convinced that the class of deterministic timed automata is closed under complementation: add a sink location, add transitions to that sink from every location, with constraints complementing the union of all the constraints labelling the outgoing transitions from that location, and finally swap final and non-final locations. As a consequence, this is not possible to construct deterministic timed automata which accept the same languages as the two timed automata of Figures 6 and 7. And it is even possible to prove that this is not possible to decide whether a timed automaton is determinizable or not [Tri03,Fin06].

Finally it is interesting to mention that the reduction to prove the undecidability of the universality problem (proof of Theorem 5) builds a non deterministic timed automaton. And indeed the universality problem (and the inclusion problem) is decidable for the class of deterministic timed automata: to check for the universality of a given deterministic timed automaton $A$, first build a (deterministic) timed automaton which accepts the complement of $L(A)$, and then check for emptiness of this automaton.

Determinizable classes of timed automata. As mentioned in the previous paragraph, not all timed automata can be determinized (i.e. there exist timed automata that accept no timed language).

\textsuperscript{20} The strictness is obvious at the syntactical level, and also holds at the semantical level, as will be argued later: there exists a timed automaton such that no deterministic timed automata accepts the same timed language.
cept timed languages which cannot be recognized by any deterministic timed automaton). However, deterministic (and hence effectively determinizable) timed automata enjoy nice closure (complementation) and decidability (universality, inclusion) properties. Verification can thus benefit of such properties.

One of the first determinizable classes of timed automata which have been investigated is the class of event-clock timed automata [AFH94]. In such an automaton every letter of the alphabet is associated two clocks, one which measures delays since the last occurrence of this action (those are called event-recording clock), and one which measures delays to the next occurrence of this action (those are called event-predicting clock). In the syntax of event-clock timed automata, resets of clocks are omitted as they are implicitly given by actions.

Example 12. In Figure 8 we give two event-clock timed automata (we take the convention that \( x_a \) is the event-recording clock associated with \( a \) whereas \( y_b \) is the event-predicting clock associated with \( b \)). In the first automaton, the time between the last \( b \) and the unique \( \ell_0 \rightarrow \ell_1 \rightarrow \ell_2 \)

\( x_a \): event-recording clock for \( a \)

\( y_b \): event-predicting clock for \( b \)

Fig. 8: Two event-clock timed automata

initial \( a \) is precisely one time unit (specified with the constraint on the last transition \( x_a = 1 \)): when we do the last \( b \), we know that the last \( a \) was precisely one time unit earlier. In the second automaton, the time between the first \( a \) and the unique final \( b \) is precisely one time unit as well (specified with the constraint on the first transition \( y_b = 1 \)): when the first \( a \) is done, we know that the next \( b \) has to be one time unit later.

We give the intuition why an event-clock timed automaton with only event-recording clocks can be determinized (the case of event-predicting clocks is more involved and we refer the reader to [AFH94] for more details). The reason is that the timed behaviour of those automata is input-determined: given a timed word, the value of the clocks after each prefix of the timed word is determined by that prefix (and not by the run followed in the timed automaton). For that reason a subset construction can be done. This kind of arguments has later been used for more complex classes of timed systems [DT04].

Recently more determinizable classes of timed automata have been investigated [BBBB09], among which we can find the class of so-called strongly non-Zeno timed automata (we omit the definition of this class here, but basically it enforces time elapsing in a rather strong way) or more dedicated classes, e.g. corresponding to logical formalisms [NP10] or to simpler classes of timed systems [SP09].
7.4 What about $\varepsilon$-transitions?

Following classical automata theory, we assume some transitions are silent, and we denote them $\varepsilon$-transitions.

*Exercise 7.* Consider the following timed automaton.

$$x = 2, a, x := 0$$

$$x = 2, \varepsilon, x := 0$$

Prove that the timed language recognized by the above timed automaton cannot be recognized by any classical timed automaton with no $\varepsilon$-transitions.