

6. Real-time models: Verifying Timed Automata

José Proença

System Verification (CC4084) 2024/2025

CISTER – U.Porto, Porto, Portugal

<https://fm-dcc.github.io/sv2425>



CISTER - Research Centre in
Real-Time & Embedded
Computing Systems

Behavioural Equivalences

Definition

A **timed trace** over a **timed LTS** is a (finite or infinite) sequence $\langle t_1, a_1 \rangle, \langle t_2, a_2 \rangle, \dots$ in $\mathcal{R}_0^+ \times Act$ such that there exists a path

$$\langle \ell_0, \eta_0 \rangle \xrightarrow{d_1} \langle \ell_0, \eta_1 \rangle \xrightarrow{a_1} \langle \ell_1, \eta_2 \rangle \xrightarrow{d_2} \langle \ell_1, \eta_3 \rangle \xrightarrow{a_2} \dots$$

such that

$$t_i = t_{i-1} + d_i$$

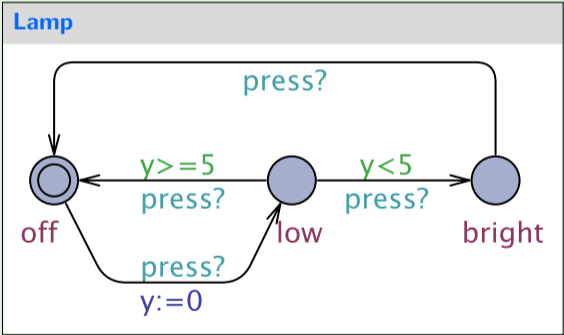
with $t_0 = 0$ and, for all clock x , $\eta_0 x = 0$.

Intuitively, each t_i is an absolute time value acting as a **time-stamp**.

Warning

All results from now on are given over an arbitrary **timed LTS**; they naturally apply to $\mathcal{T}(ta)$ for any timed automata ta .

Ex. 6.1: Write 4 possible timed traces

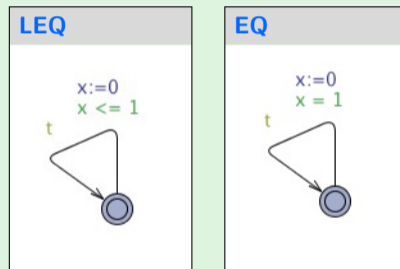


Given a **timed trace** tc , the corresponding **untimed trace** is $(\pi_2)^\omega tc$.

Definition

- two states s_1 and s_2 of a timed LTS are **timed-language equivalent** if the **set of finite timed traces** of s_1 and s_2 coincide;
- ... similar definition for **untimed-language equivalent** ...

Ex. 6.2: Why?



are not **timed-language equivalent**

Timed bisimulation (between states of timed LTS)

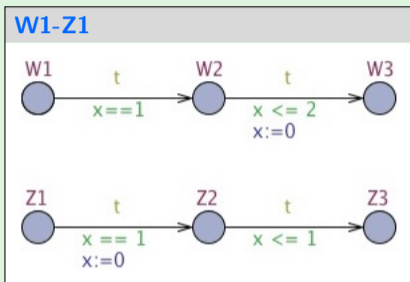
A relation R is a **timed simulation** iff whenever $s_1 R s_2$, for any action a and delay d ,

$$s_1 \xrightarrow{a} s'_1 \Rightarrow \text{there is a transition } s_2 \xrightarrow{a} s'_2 \wedge s'_1 R s'_2$$

$$s_1 \xrightarrow{d} s'_1 \Rightarrow \text{there is a transition } s_2 \xrightarrow{d} s'_2 \wedge s'_1 R s'_2$$

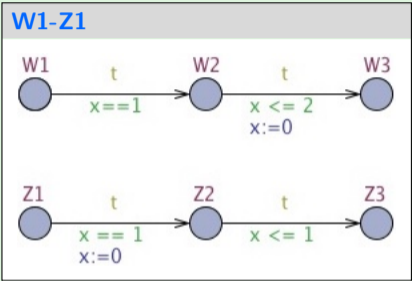
And a **timed bisimulation** if its converse is also a timed simulation.

Example



W1 bisimilar to Z1?

Example



W1 bisimilar to Z1?

$$\langle\langle W1, \{x \mapsto 0\} \rangle, \langle Z1, \{x \mapsto 0\} \rangle\rangle \in R$$

where

$$\begin{aligned}
 R = & \{ \langle \langle W1, \{x \mapsto d\} \rangle, \langle Z1, \{x \mapsto d\} \rangle \rangle \mid d \in \mathcal{R}_0^+ \} \cup \\
 & \{ \langle \langle W2, \{x \mapsto d + 1\} \rangle, \langle Z2, \{x \mapsto d\} \rangle \rangle \mid d \in \mathcal{R}_0^+ \} \cup \\
 & \{ \langle \langle W3, \{x \mapsto d\} \rangle, \langle Z3, \{x \mapsto e\} \rangle \rangle \mid d, e \in \mathcal{R}_0^+ \}
 \end{aligned}$$

Untimed bisimulation

A relation R is an **untimed simulation** iff whenever $s_1 R s_2$, for any action a and delay t ,

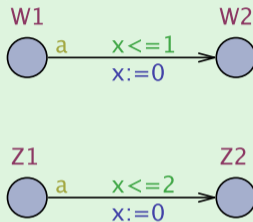
$$s_1 \xrightarrow{a} s'_1 \Rightarrow \text{there is a transition } s_2 \xrightarrow{a} s'_2 \wedge s'_1 R s'_2$$

$$s_1 \xrightarrow{d} s'_1 \Rightarrow \text{there is a transition } s_2 \xrightarrow{d'} s'_2 \wedge s'_1 R s'_2$$

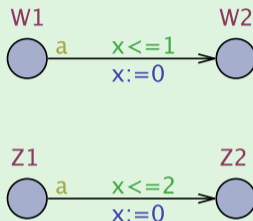
And it is an **untimed bisimulation** if its converse is also an untimed simulation.

Alternatively, it can be defined over a modified LTS in which all delays are abstracted on a unique, special transition labelled by ϵ .

Ex. 6.3: W1 bisimilar to Z1?



Ex. 6.3: W1 bisimilar to Z1?



$$\langle\langle W1, \{x \mapsto 0\} \rangle, \langle Z1, \{x \mapsto 0\} \rangle \rangle \in R$$

where

$$R = \{ \langle\langle W1, \{x \mapsto d\} \rangle, \langle Z1, \{x \mapsto d'\} \rangle \rangle \mid 0 \leq d \leq 1, 0 \leq d' \leq 2 \} \cup \dots$$

Behavioural Properties

The satisfaction problem

Given a **timed automata** ta and a **property** ϕ , show that

$$\mathcal{T}(ta) \models \phi$$

The satisfaction problem

Given a **timed automata** ta and a **property** ϕ , show that

$$\mathcal{T}(ta) \models \phi$$

- in which logic language shall ϕ be specified?
- how is \models defined?

Uppaal variant of CTL

- **state formulae**: describes individual states in $\mathcal{T}(ta)$
- **path formulae**: describes properties of paths in $\mathcal{T}(ta)$

State formulae

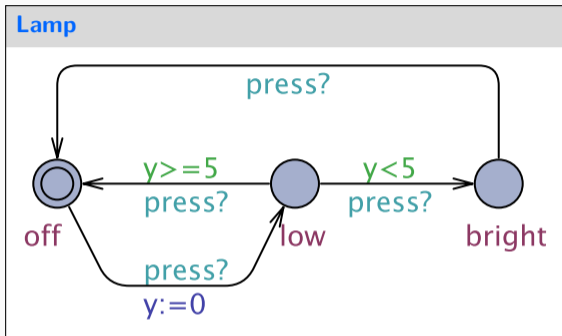
$$\Psi ::= ta.l \mid g_c \mid g_d \mid \text{deadlock} \mid \text{not } \Psi \mid \Psi \text{ or } \Psi \mid \Psi \text{ and } \Psi \mid \Psi \text{ imply } \Psi$$

Any expression which can be evaluated to a boolean value for a state (typically involving the **clock constraints** used for guards and invariants and similar constraints over integer variables):

$$x \geq 8, i == 8 \text{ and } x < 2, \dots$$

Additionally,

- **ta.l** which tests **current location**: $(l, \eta) \models ta.l$
provided (l, η) is a state in $\mathcal{T}(ta)$
- **deadlock**: $(l, \eta) \models \forall_{d \in \mathcal{R}_0^+}. \text{there is no transition from } \langle l, \eta + d \rangle$



Ex. 6.4: Write a state formula

1. The lamp is low
2. Not off and $y > 25$
3. If it is low or bright, then $y \leq 3600$

Path formulae

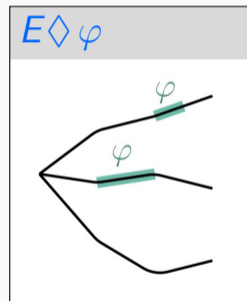
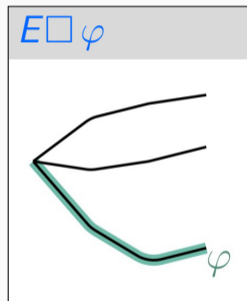
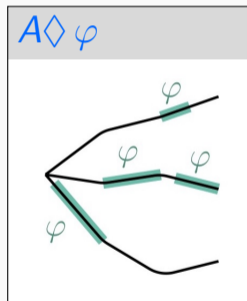
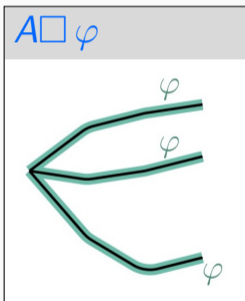
$$\Pi ::= A\Box\Psi \mid A\Diamond\Psi \mid E\Box\Psi \mid E\Diamond\Psi \mid \Phi \rightsquigarrow \Psi$$

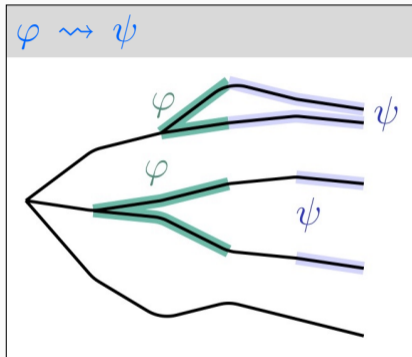
where

- A, E quantify (universally and existentially, resp.) over **paths**
- \Box, \Diamond quantify (universally and existentially, resp.) over **states in a path**

also notice that

$$\Phi \rightsquigarrow \Psi \stackrel{\text{abv}}{=} A\Box(\Phi \Rightarrow A\Diamond\Psi)$$





Example

If a message is sent, it will eventually be received – $\text{send}(m) \rightsquigarrow \text{received}(m)$

$E\Diamond\phi$

Is there a path starting at the initial state, such that a state formula ϕ is eventually satisfied?

- Often used to perform sanity checks on a model:
 - is it possible for a sender to send a message?
 - can a message possibly be received?
 - ...
- Do not by themselves guarantee the correctness of the protocol (i.e. **that any message is eventually delivered**), but they validate the basic behavior of the model.

$A\Box\phi$ **and** $E\Box\phi$

Something bad will never happen
or something bad will possibly never happen

Examples

- In a nuclear power plant the temperature of the core is always (invariantly) under a certain threshold.
- In a game a safe state is one in which we can still win, ie, will possibly not loose.

In Uppaal these properties are formulated positively: something good is invariantly true.

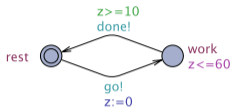
$A \diamond \phi$ and $\phi \rightsquigarrow \psi$

Something good will *eventually happen*
or if *something* happens, then *something else* will eventually happen!

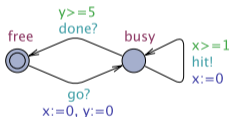
Examples

- When pressing the on button, then eventually the television should turn on.
- In a communication protocol, any message that has been sent should eventually be received.

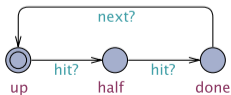
Worker



Hammer



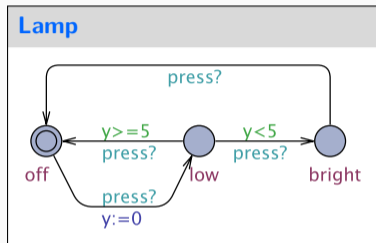
Nail



Ex. 6.5: Write properties and explain them

1. Using $E\Diamond$
2. Using $E\Box$
3. Using $A\Diamond$
4. Using $A\Box$
5. Using \rightsquigarrow

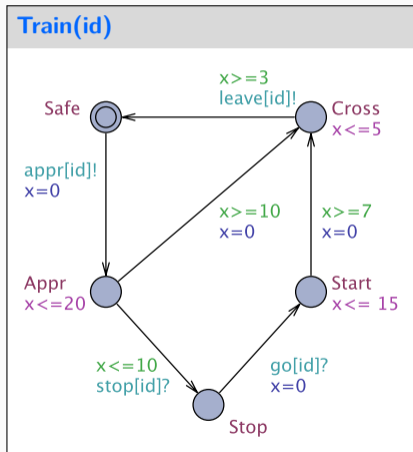
(Practice in UPPAAL)



Ex. 6.6: Write formulas, and say which ones are true

1. The lamp can become bright;
2. The lamp will eventually become bright;
3. The lamp can never be on for more than 3600s;
4. It is possible to never turn on the lamp;
5. Whenever the light is bright, the clock y is non-zero;
6. Whenever the light is bright, it will eventually become off.

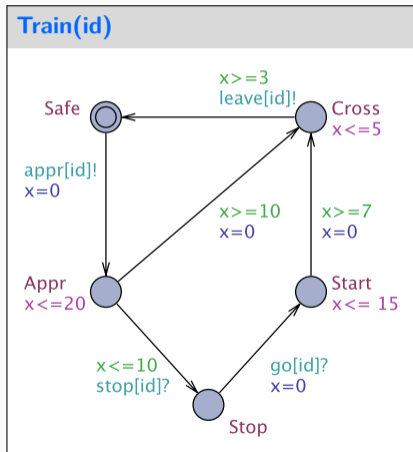
Examples: proving mutual exclusion



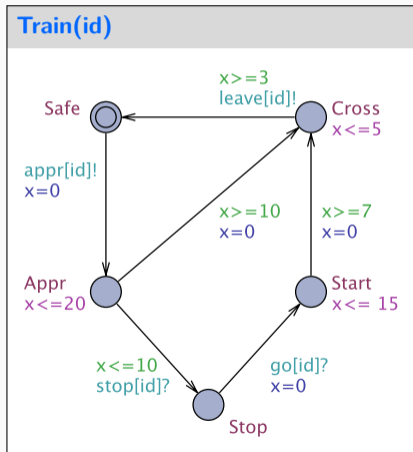
(Train 0 can reach the cross)

(Train 0 can be crossing bridge while Train 1 is waiting to cross)

(Train 0 can cross bridge while the other trains are waiting to cross)



- $E \langle \rangle Train(0).Cross$
(Train 0 can reach the cross)
- $E \langle \rangle Train(0).Cross$ and $Train(1).Stop$
(Train 0 can be crossing bridge while Train 1 is waiting to cross)
- $E \langle \rangle Train(0).Cross$ and
(forall $(i:id-t)$
 $i \neq 0$ imply $Train(i).Stop$)
(Train 0 can cross bridge while the other trains are waiting to cross)



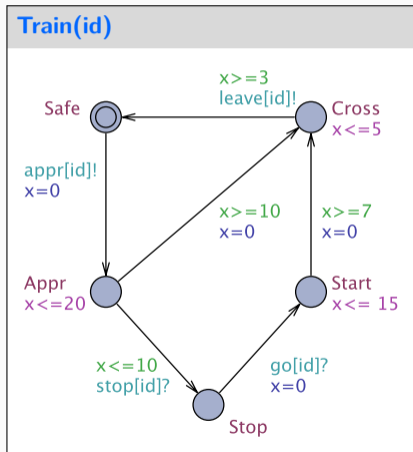
There can never be N elements in the queue

There is never more than one train crossing the bridge

Whenever a train approaches the bridge, it will eventually cross

The system is deadlock-free

The train gate example (2/2)



- $A[] \text{ Gate.list}[N] == 0$
There can never be N elements in the queue
- $A[] \text{ forall } (i:id-t) \text{ forall } (j:id-t) \text{ Train}(i).\text{Cross} \ \&\& \ \text{Train}(j).\text{Cross} \ \text{imply } i == j$
There is never more than one train crossing the bridge
- $\text{Train}(1).\text{Appr} \rightarrow \text{Train}(1).\text{Cross}$
Whenever a train approaches the bridge, it will eventually cross
- $A[] \text{ not deadlock}$
The system is deadlock-free

Properties

- **mutual exclusion**: no two processes are in their critical sections at the same time
- **deadlock freedom**: if some process is trying to access its critical section, then eventually some process (not necessarily the same) will be in its critical section; similarly for exiting the critical section

The Problem

- Dijkstra's original asynchronous algorithm (1965) requires, for n processes to be controlled, $\mathcal{O}(n)$ read-write registers and $\mathcal{O}(n)$ operations.
- This result is a theoretical limit (proved by Lynch and Shavit in 1992) which compromises scalability.

The Problem

- Dijkstra's original asynchronous algorithm (1965) requires, for n processes to be controlled, $\mathcal{O}(n)$ read-write registers and $\mathcal{O}(n)$ operations.
- This result is a theoretical limit (proved by Lynch and Shavit in 1992) which compromises scalability.

but it can be overcome by introducing specific **timing constraints**

Two *timed* algorithms:

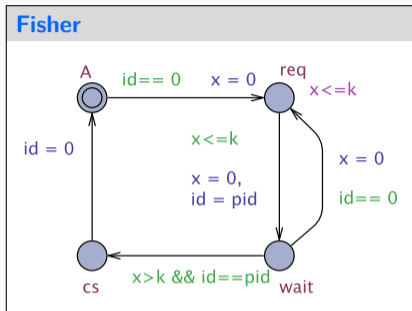
- **Fisher's protocol** (included in the UPPAAL distribution)
- **Lamport's protocol**

The algorithm

```
repeat
  repeat
    await  $id = 0$ 
     $id := i$ 
    delay( $k$ )
  until  $id = i$ 
  (critical section)
   $id := 0$ 
forever
```

Comments

- One shared read/write register (the variable id)
- Behaviour depends crucially on the value for k — the **time delay**
- Constant k should be **larger than the longest time that a process may take to perform a step while trying to get access to its critical section**
- This choice guarantees that whenever process i finds $id = i$ on testing the loop guard it can enter safely its critical section: **all** other processes are out of the loop or with their index in id overwritten by i .



- Each process uses a local clock x to guarantee that the upper bound between its successive steps, while trying to access the critical section, is k (cf. **invariant** in state *req*).
- Invariant** in state *req* establishes k as such an upper bound
- Guard** in transition from *wait* to *cs* ensures the correct delay before entering the critical section

Properties

```
% P(1) requests access => it will eventually wait
P(1).req --> P(1).wait
% the algorithm is deadlock-free
A[] not deadlock
% mutual exclusion invariant
A[] forall (i:int[1,6]) forall (j:int[1,6])
  P(i).cs && P(j).cs imply i == j
```

- The algorithm is **deadlock-free**
- It ensures mutual exclusion if the correct timing constraints.
- ... but it is critically sensible to small violations of such constraints: for example, replacing $x > k$ by $x \geq k$ in the transition leading to *cs* compromises both **mutual exclusion** and **liveness**.

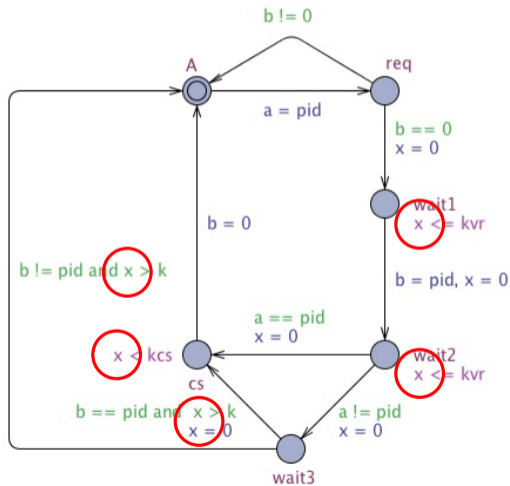
The algorithm

```
start :  $a := i$   
       if  $b \neq 0$  then goto start  
        $b := i$   
       if  $a \neq i$  then delay( $k$ )  
           else if  $b \neq i$  then goto start  
       (critical section)  
        $b := 0$ 
```

Comments

- Two shared read/write registers (variables a and b)
- Avoids **forced waiting** when no other processes are requiring access to their critical sections

Lamport(pid)



Model time constants:

- k — time delay
- kvr — max bound for register access
- kcs — max bound for permanence in critical section

Typically $k \geq kvr + kcs$

Experiments

	k	kvr	kcs	verified?
Mutual Exclusion	4	1	1	Yes
Mutual Exclusion	2	1	1	Yes
Mutual Exclusion	1	1	1	No
No deadlock	4	1	1	Yes
No deadlock	2	1	1	Yes
No deadlock	1	1	1	Yes