Time-critical reactive systems (verification)

José Proença

HASLab - INESC TEC
Universidade do Minho
Braga, Portugal

May 2016

Traces

Definition

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

$$\langle l_0, \eta_0\rangle \xrightarrow{d_1} \langle l_0, \eta_1\rangle \xrightarrow{a_1} \langle l_1, \eta_2\rangle \xrightarrow{d_2} \langle l_1, \eta_3\rangle \xrightarrow{a_2} \cdots$$

such that

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

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

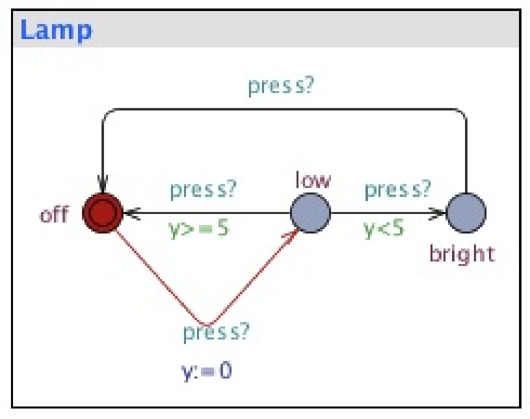
Intuitively, each t_i is an absolute time value acting as a time-stamp.

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.

Traces

Write possible traces



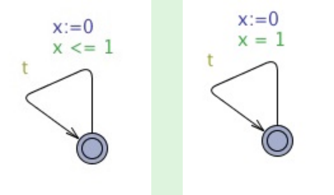
Traces

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 ...

Example



are not timed-language equivalent

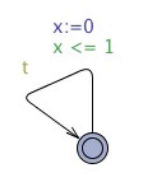
Traces

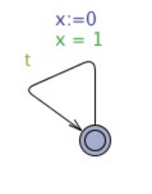
Given a timed trace tc, the corresponding untimed trace is $(\pi_2)^\omega\ tc$.

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 ...

Example





are not timed-language equivalent

$\langle(0, t)\rangle$ is not a trace of the TLTS generated by the second system.

Bisimulation

Timed bisimulation (between states of timed LTS)

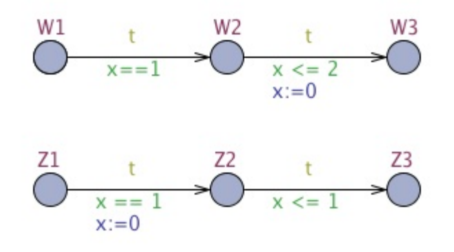
A relation R is a timed simulation iff whenever $s_1 R s_2$, for any action a and delay d,

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

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

Bisimulation

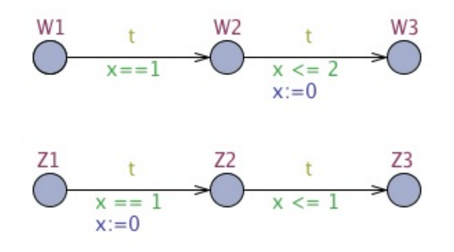
Example



W1 bisimilar to Z1?

Bisimulation

Example



W1 bisimilar to Z1?

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

where

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

Untimed Bisimulation

Untimed bisimulation

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

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

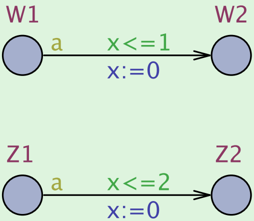
$$s_1 \xrightarrow{d} s_1' \Rightarrow \text{there is a transition} \quad s_2 \xrightarrow{d'} s_2' \wedge s_1' R s_2'$$

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.

Untimed Bisimulation

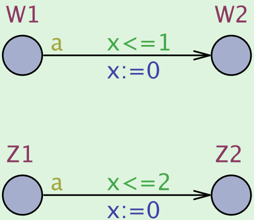
Example



W1 bisimilar to Z1?

Untimed Bisimulation

Example



W1 bisimilar to Z1?

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

where

$$\begin{aligned} R \;=\; & \{\langle\langle W1, \{x \mapsto d\}\rangle\ , \langle Z1, \{x \mapsto d'\}\rangle\rangle \mid 0 \leq d \leq 1, 0 \leq d' \leq 2\} \;\cup \\ & \{\langle\langle W1, \{x \mapsto d\}\rangle\ , \langle Z1, \{x \mapsto d'\}\rangle\rangle \mid d > 1, d' > 2\} \;\cup \\ & \{\langle\langle W2, \{x \mapsto d\}\rangle\ , \langle Z2, \{x \mapsto d'\}\rangle\rangle \mid d, d' \in \mathcal{R}_0^+\} \end{aligned}$$

Properties: expression and satisfaction

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?

Properties: expression and satisfaction

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?

Expressing properties: Uppaal

Uppaal variant of Ctl

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

Expressing properties: Uppaal

State formulae

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 >= 8`, `i == 8 and x < 2`, ...

Additionally,

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

Expressing properties: Uppaal

Path formulae

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

$$\Psi \ ::= ta.\ell \mid g_c \mid g_d \mid \texttt{not}\ \Psi \mid \Psi\ \texttt{or}\ \Psi \mid \Psi\ \texttt{and}\ \Psi \mid \Psi\ \texttt{imply}\ \Psi$$

where

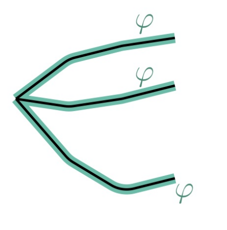
- A, E quantify (universally and existentially, resp.) over paths
- $\Box$, $\Diamond$ quantify (universally and existentially, resp.) over states in a path

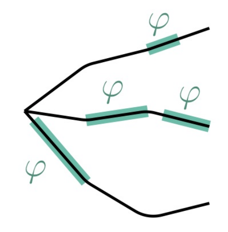
also notice that

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

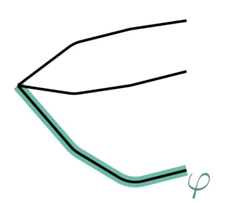
Expressing properties: Uppaal

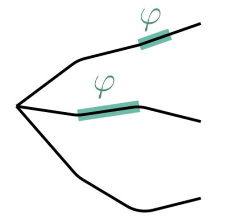
$A\Box\,\varphi$ and $A\Diamond\,\varphi$





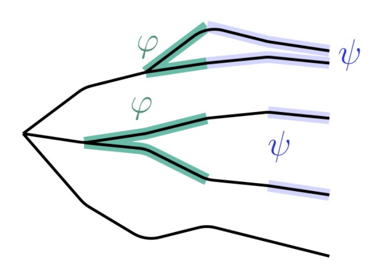
$E\Box\,\varphi$ and $E\Diamond\,\varphi$





Expressing properties: Uppaal

$\varphi \rightsquigarrow \psi$



Example

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

Reachability properties

$E\Diamond\,\phi$

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.

Safety properties

$A\Box\,\phi$ 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.

Liveness properties

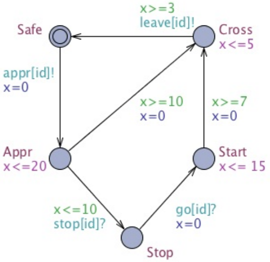
$A\Diamond\,\phi$ and $\phi \leadsto \psi$

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

Examples

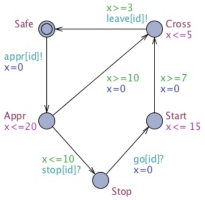
- When pressing the on button, then eventually the television should turn on.
- In a communication protocol, any message that has been sent should eventually be received.

The train gate example



- `E<> Train(0).Cross`
  (Train 0 can reach the cross)
- `E<> Train(0).Cross and Train(1).Stop`
  (Train 0 can be crossing bridge while Train 1 is waiting to cross)
- ```
  E<> Train(0).Cross and
          (forall (i:id-t) i != 0 imply Train(i).Stop)
  ```
  (Train 0 can cross bridge while the other trains are waiting to cross)
  ```

The train gate example



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

Mutual exclusion

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

Mutual exclusion

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

Mutual exclusion

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

Fisher's algorithm

The algorithm

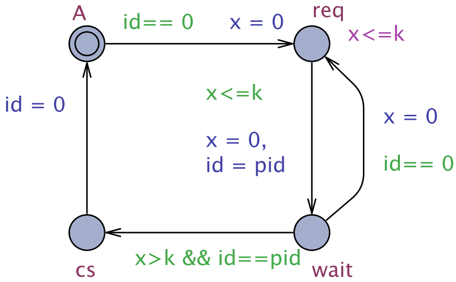
repeat
    repeat
        await $id = 0$
        $id := i$
        delay(k)
    until $id = i$
    (critical section)
    $id := 0$
forever

Fisher's algorithm

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 ist critical section: all other processes are out of the loop or with their index in *id* overwritten by i.

Fisher's algorithm in Uppaal



- Each process uses a local clock *x* to guarantee that the upper bound between 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

Fisher's algorithm in Uppaal

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.

Lamport's algorithm

The algorithm

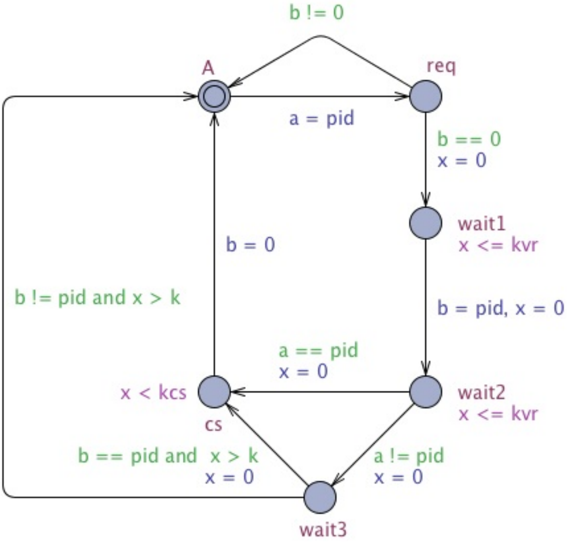
$$
\begin{aligned}
\text{start}: \quad & a := i \\
& \text{if } b \neq 0 \text{ then goto start} \\
& b := i \\
& \text{if } a \neq i \text{ then delay}(k) \\
& \qquad\qquad \text{else if } b \neq i \text{ then goto start} \\
& \textit{(critical section)} \\
& b := 0
\end{aligned}
$$

Lamport's algorithm

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's algorithm in Uppaal



Lamport's algorithm

Model time constants:

k — time delay

kvr — max bound for register access

kcs — max bound for permanence in critical section

Typically

$$\boxed{k \geq kvr + kcs}$$

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