

Time-critical reactive systems (modelling)

José Proença

MAPi 2019/20

October 2019

HASLab/INESC-TEC & CISTER/ISEP, Portugal

Motivation

Motivation

Specifying an airbag saying that **in a car crash the airbag eventually inflates** maybe not enough, but:

in a car crash the airbag eventually inflates **within 20ms**

*Correctness in time-critical systems not only depends on the logical result of the computation, but also **on the time at which the results are produced***

[Baier & Katoen, 2008]

Examples of time-critical systems

Lip-synchronization protocol

Synchronizes the separate video and audio sources bounding on the amount of time mediating the presentation of a video frame and the corresponding audio frame. Humans tolerate less than 160 ms.

Bounded retransmission protocol

Controls communication of large files over infrared channel between a remote control unit and a video/audio equipment. Correctness depends crucially on

- transmission and synchronization delays
- time-out values for times at sender and receiver

And many others...

- medical instruments
- hybrid systems (eg for controlling industrial plants)

Course structure

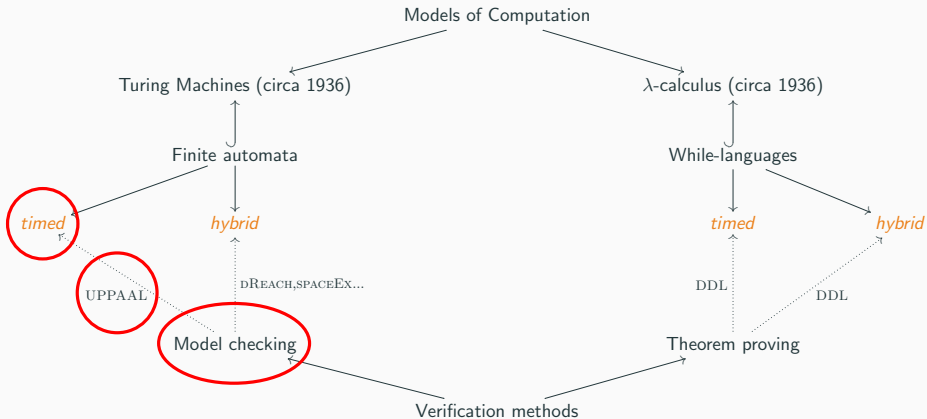


Table of contents

1. Motivation
2. Timed Automata
3. Semantics
4. Modelling in UPPAAL
5. Behavioural Equivalences
6. Behavioural Properties
7. Examples: proving mutual exclusion

- **timed transition systems, timed Petri nets, timed IO automata, timed process algebras** and other formalisms associate lower and upper bounds to transitions, but no **time constraints** to transverse the automaton.
- Expressive power is often somehow limited and **infinite**-state LTS (introduced to express **dense** time models) are difficult to handle in practice

Example

Typical process algebra tools are unable to express a system which has one action a which can only occur at time point 5 with the effect of moving the system to its initial state.

This example has, however, a simple description in terms of time measured by a stopwatch:

1. Set the stopwatch to 0
2. When the stopwatch measures 5, action a can occur. If a occurs go to 1., if not idle forever.

Motivation

This suggests resorting to an **automaton-based formalism** with an explicit notion of **clock** (stopwatch) to control availability of transitions.

Timed Automata [Alur & Dill, 90]

- emphasis on decidability of the model-checking problem and corresponding practically efficient algorithms
- infinite underlying timed transition systems are converted to **finitely large** symbolic transition systems where **reachability** becomes decidable (**region** or **zone** graphs)

Associated tools

- UPPAAL [Behrmann, David, Larsen, 04]
- KRONOS [Bozga, 98]

UPPAAL = (Uppsala University + Aalborg University) [1995]

- A toolbox for modeling, simulation and verification of real-time systems
- where systems are modeled as networks of timed automata enriched with integer variables, structured data types, channel synchronisations and urgency annotations
- Properties are specified in a subset of CTL

www.uppaal.com

Timed Automata

Timed automata

Finite-state machine equipped with a finite set of real-valued clock variables (**clocks**)

Clocks

- **dense-time** model
- clocks can only be **inspected** or
- **reset to zero**, after which they start increasing their value implicitly as time progresses
- the value of a clock corresponds to time elapsed since its last reset
- all clocks proceed synchronously (at the same rate)

Definition

$$\langle L, L_0, Act, C, Tr, Inv \rangle$$

where

- L is a set of **locations**, and $L_0 \subseteq L$ the set of **initial** locations
- Act is a set of **actions** and C a set of **clocks**
- $Tr \subseteq L \times \mathcal{C}(C) \times Act \times \mathcal{P}(C) \times L$ is the **transition relation**

$$l_1 \xrightarrow{g, a, U} l_2$$

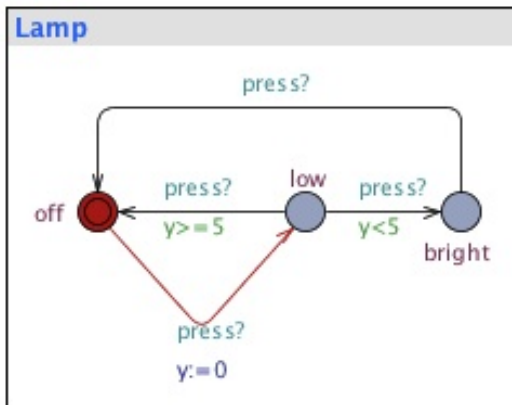
denotes a transition from location l_1 to l_2 , **labelled** by a , enabled if **guard** g is valid, which, when performed, **resets** the set U of **clocks**

- $Inv : L \rightarrow \mathcal{C}(C)$ is the assignment of **invariants** to locations

where $\mathcal{C}(C)$ denotes the set of clock constraints over a set C of clock variables

Example: the lamp interrupt

(extracted from UPPAAL)



Clock constraints

$\mathcal{C}(C)$ denotes the set of clock constraints over a set C of clock variables.
Each constraint is formed according to

$$g ::= x \square n \mid x - y \square n \mid g \wedge g \mid true$$

where $x, y \in C, n \in \mathbb{N}$ and $\square \in \{<, \leq, >, \geq, =\}$

used in

- **transitions** as **guards** (enabling conditions)

a transition cannot occur if its guard is invalid

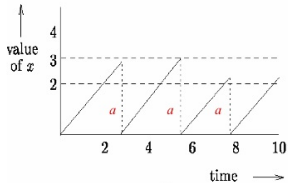
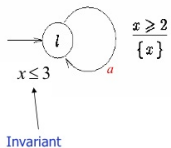
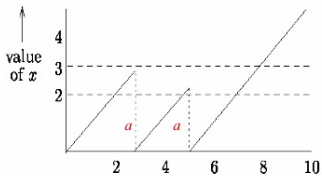
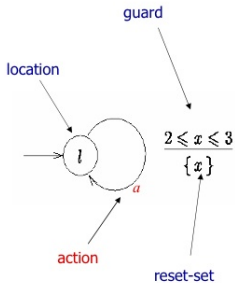
- **locations** as **invariants** (safety specifications)

a location must be left before its invariant becomes invalid

Note

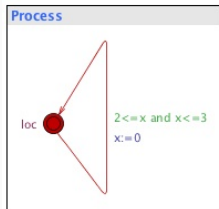
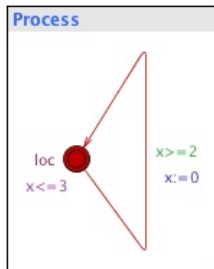
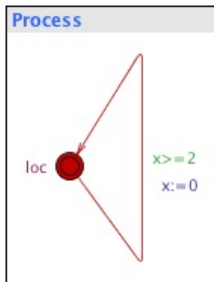
Invariants are the **only** way to force transitions to occur

Guards, updates & invariants



Transition guards & location invariants

Demo (in Uppaal)



Parallel composition of timed automata

- Action labels as **channel** identifiers
- Communication by **forced handshaking** over a subset of common actions
- Can be defined as an associative binary operator (as in the tradition of process algebra) or as an automaton construction over a finite set of timed automata originating a so-called **network** of timed automata

Parallel composition of timed automata

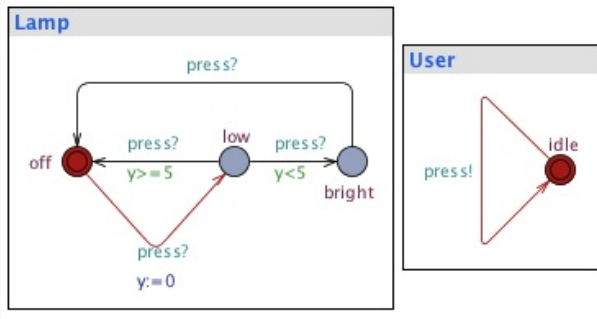
Let $H \subseteq Act_1 \cap Act_2$. The parallel composition of ta_1 and ta_2 synchronizing on H is the timed automata

$$ta_1 \parallel_H ta_2 := \langle L_1 \times L_2, L_{0,1} \times L_{0,2}, Act_{\parallel_H}, C_1 \cup C_2, Tr_{\parallel_H}, Inv_{\parallel_H} \rangle$$

where

- $Act_{\parallel_H} = ((Act_1 \cup Act_2) - H) \cup \{\tau\}$
- $Inv_{\parallel_H} \langle l_1, l_2 \rangle = Inv_1(l_1) \wedge Inv_2(l_2)$
- Tr_{\parallel_H} is given by:
 - $\langle l_1, l_2 \rangle \xrightarrow{g,a,U} \langle l'_1, l_2 \rangle$ if $a \notin H \wedge l_1 \xrightarrow{g,a,U} l'_1$
 - $\langle l_1, l_2 \rangle \xrightarrow{g,a,U} \langle l_1, l'_2 \rangle$ if $a \notin H \wedge l_2 \xrightarrow{g,a,U} l'_2$
 - $\langle l_1, l_2 \rangle \xrightarrow{g,\tau,U} \langle l'_1, l'_2 \rangle$ if $a \in H \wedge l_1 \xrightarrow{g_1,a,U_1} l'_1 \wedge l_2 \xrightarrow{g_2,a,U_2} l'_2$
with $g = g_1 \wedge g_2$ and $U = U_1 \cup U_2$

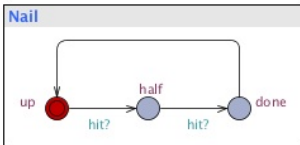
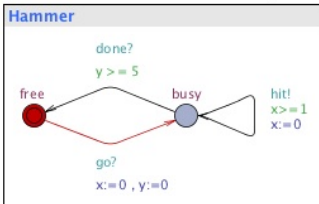
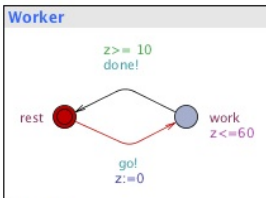
Example: the lamp interrupt as a closed system



Uppaal:

- takes $H = Act_1 \cap Act_2$ (actually as **complementary** actions denoted by the **?** and **!** annotations)
- only deals with **closed** systems

Exercise: worker, hammer, nail



Semantics

Timed Labelled Transition Systems

Syntax

Semantics

How to write

How to execute

Timed Automaton

TLTS (Timed LTS)

Timed Labelled Transition Systems

Syntax	Semantics
<i>How to write</i>	<i>How to execute</i>
Timed Automaton	TLTS (Timed LTS)

Timed LTS

Introduce **delay transitions** to capture the passage of time within a LTS:

$s \xrightarrow{a} s'$ for $a \in Act$, are ordinary transitions due to action occurrence

$s \xrightarrow{d} s'$ for $d \in \mathcal{R}^+$, are **delay** transitions

subject to a number of constraints, eg,

Timed LTS

- time additivity

$$(s \xrightarrow{d} s' \wedge 0 \leq d' \leq d) \Rightarrow s \xrightarrow{d'} s'' \xrightarrow{d-d'} s' \text{ for some state } s''$$

- delay transitions are deterministic

$$(s \xrightarrow{d} s' \wedge s \xrightarrow{d} s'') \Rightarrow s' = s''$$

Semantics of TA:

Every TA ta defines a TLTS

$\mathcal{T}(ta)$

whose states are pairs

$\langle \text{location, clock valuation} \rangle$

with **infinitely**, even **uncountably** many states, and infinite branching

Clock valuations

Definition

A **clock valuation** η for a set of clocks C is a function

$$\eta : C \longrightarrow \mathcal{R}_0^+$$

assigning to each clock $x \in C$ its current value ηx .

Satisfaction of clock constraints

$$\eta \models x \square n \Leftrightarrow \eta x \square n$$

$$\eta \models x - y \square n \Leftrightarrow (\eta x - \eta y) \square n$$

$$\eta \models g_1 \wedge g_2 \Leftrightarrow \eta \models g_1 \wedge \eta \models g_2$$

Operations on clock valuations

Delay

For each $d \in \mathcal{R}_0^+$, valuation $\eta + d$ is given by

$$(\eta + d)x = \eta x + d$$

Reset

For each $R \subseteq C$, valuation $\eta[R]$ is given by

$$\begin{cases} \eta[R]x = \eta x & \Leftarrow x \notin R \\ \eta[R]x = 0 & \Leftarrow x \in R \end{cases}$$

From ta to $\mathcal{T}(ta)$

Let $ta = \langle L, L_0, Act, C, Tr, Inv \rangle$

$$\mathcal{T}(ta) = \langle S, S_0 \subseteq S, N, T \rangle$$

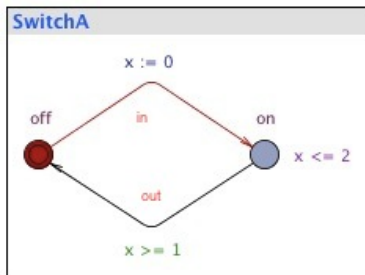
where

- $S = \{ \langle l, \eta \rangle \in L \times (\mathcal{R}_0^+)^C \mid \eta \models Inv(l) \}$
- $S_0 = \{ \langle \ell_0, \eta \rangle \mid \ell_0 \in L_0 \wedge \eta x = 0 \text{ for all } x \in C \}$
- $N = Act \cup \mathcal{R}_0^+$ (ie, transitions can be labelled by actions or delays)
- $T \subseteq S \times N \times S$ is given by:

$$\langle l, \eta \rangle \xrightarrow{a} \langle l', \eta' \rangle \iff \exists_{l' \xrightarrow{g, a, U} l' \in Tr} \eta \models g \wedge \eta' = \eta[U] \wedge \eta' \models Inv(l')$$

$$\langle l, \eta \rangle \xrightarrow{d} \langle l, \eta + d \rangle \iff \exists_{d \in \mathcal{R}_0^+} \eta + d \models Inv(l)$$

Example: the simple switch

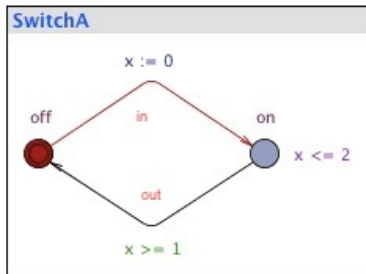


$\mathcal{T}(\text{SwitchA})$

$$S = \{\langle \text{off}, t \rangle \mid t \in \mathcal{R}_0^+\} \cup \{\langle \text{on}, t \rangle \mid 0 \leq t \leq 2\}$$

where t is a shorthand for η such that $\eta x = t$

Example: the simple switch



$\mathcal{T}(\text{SwitchA})$

$$\langle \text{off}, t \rangle \xrightarrow{d} \langle \text{off}, t + d \rangle \text{ for all } t, d \geq 0$$

$$\langle \text{off}, t \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \text{ for all } t \geq 0$$

$$\langle \text{on}, t \rangle \xrightarrow{d} \langle \text{on}, t + d \rangle \text{ for all } t, d \geq 0 \text{ and } t + d \leq 2$$

$$\langle \text{on}, t \rangle \xrightarrow{\text{out}} \langle \text{off}, t \rangle \text{ for all } 1 \leq t \leq 2$$

- The elapse of time in timed automata **only** takes place in locations:
- ... actions take place instantaneously
- Thus, several actions may take place at a single time unit

Behaviours

- Paths in $\mathcal{T}(ta)$ are discrete representations of continuous-time behaviours in ta
- ... at least they indicate the states immediately before and after the execution of an action
- However, as interval delays may be realised in uncountably many different ways, different paths may represent the same behaviour

Behaviours

- Paths in $\mathcal{T}(ta)$ are **discrete representations of continuous-time behaviours** in ta
- ... at least they indicate the states immediately before and after the execution of an action
- However, as interval delays may be realised in **uncountably** many different ways, different paths may represent the same behaviour
- ... but not all paths correspond to valid (**realistic**) behaviours:

undesirable paths:

- **time-convergent** paths
- **timelock** paths
- **zeno** paths

Time-convergent paths

$$\langle l, \eta \rangle \xrightarrow{d_1} \langle l, \eta + d_1 \rangle \xrightarrow{d_2} \langle l, \eta + d_1 + d_2 \rangle \xrightarrow{d_3} \langle l, \eta + d_1 + d_2 + d_3 \rangle \xrightarrow{d_4} \dots$$

such that

$$\forall i \in \mathbb{N}. d_i > 0 \wedge \sum_{i \in \mathbb{N}} d_i = d$$

ie, the infinite sequence of delays converges toward d

- Time-convergent paths are **conterintuitive**; as their existence cannot be avoided, they are simply **ignored** in the semantics of Timed Automata
- Time-**divergent** paths are the ones in which time always progresses

Time-convergent paths

Definition

An infinite path fragment ρ is **time-divergent** if $\text{ExecTime}(\rho) = \infty$
Otherwise is **time-convergent**.

where

$$\text{ExecTime}(\rho) = \sum_{i=0.. \infty} \text{ExecTime}(\delta_i)$$

$$\text{ExecTime}(\delta) = \begin{cases} 0 & \Leftarrow \delta \in \text{Act} \\ \delta & \Leftarrow \delta \in \mathcal{R}_0^+ \end{cases}$$

for ρ a path and δ a label in $\mathcal{T}(ta)$

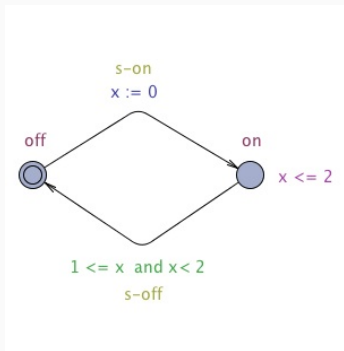
Definition

A path is **timelock** if it contains a state with a timelock, ie, a **state from which there is not any time-divergent path**

A **timelock** represents a situation that causes time progress to halt (e.g. when it is impossible to leave a location before its invariant becomes invalid)

- any **terminal state** (\neq terminal location) in $\mathcal{T}(ta)$ contains a timelock
- ... but not all timelocks arise as terminal states in $\mathcal{T}(ta)$

Timelock paths

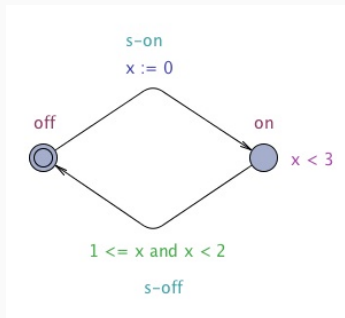


State $\langle on, 2 \rangle$ is reachable through path

$$\langle off, 0 \rangle \xrightarrow{s-on} \langle on, 0 \rangle \xrightarrow{2} \langle on, 2 \rangle$$

and is terminal

Timelock paths



State $\langle on, 2 \rangle$ is not terminal but has a **convergent** path:

$\langle on, 2 \rangle \langle on, 2.9 \rangle \langle on, 2.99 \rangle \langle on, 2.999 \rangle \dots$

In a Timed Automaton

- The elapse of time only takes place at **locations**
- Actions occur **instantaneously**: at a single time instant several actions may take place

... it may perform **infinitely** many actions in a **finite** time interval
(non realizable because it would require infinitely fast processors)

In a Timed Automaton

- The elapse of time only takes place at **locations**
- Actions occur **instantaneously**: at a single time instant several actions may take place

... it may perform **infinitely** many actions in a **finite** time interval
(non realizable because it would require infinitely fast processors)

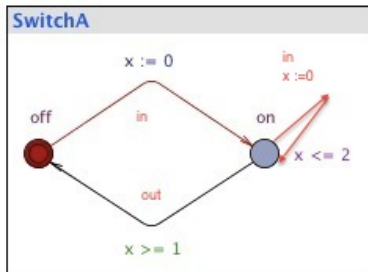
Definition

An infinite path fragment ρ is **zeno** if it is **time-convergent** and **infinitely many actions occur along it**

A timed automaton ta is **non-zeno** if there is not an initial zeno path in $\mathcal{T}(ta)$

Example

Suppose the user can press the *in* button when the light is *on* in



In doing so clock x is reset to 0 and light stays *on* for more 2 time units (unless the button is pushed again ...)

Example

Typical paths: The user presses *in* infinitely fast:

$$\langle \text{off}, 0 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{\text{in}} \dots$$

The user presses *in* faster and faster:

$$\langle \text{off}, 0 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{0.5} \langle \text{on}, 0.5 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{0.25} \langle \text{on}, 0.25 \rangle \xrightarrow{\text{in}} \langle \text{on}, 0 \rangle \xrightarrow{0.125} \dots$$

How can this be fixed?

Time shall pass!

Sufficient criterion for nonzenoness

A timed automaton is nonzeno if on any of its control cycles time advances with at least some **constant amount** (≥ 0). Formally, if for every control cycle

$$\ell_0 \xrightarrow{g_0, a_0, U_0} \ell_1 \xrightarrow{g_1, a_1, U_1} \dots \xrightarrow{g_n, a_n, U_n} \ell_0$$

there exists a clock $x \in C$ such that

1. $x \in U_i$ (for $0 \leq i \leq n$)
2. for all clock valuations η , there is a $c \in \mathbb{N}_{>0}$ such that

$$\eta(x) < c \Rightarrow ((\eta \not\models g_j) \vee \neg \text{Inv}(\ell_j)) \text{ for some } 0 \leq j \leq n$$

Both

- timelocks
- zenoness

are **modelling flaws** and need to be avoided.

Example

In the example above, it is enough to impose a non zero minimal delay between successive button pushings.

Modelling in Uppaal

... an editor, simulator and model-checker for TA with extensions ...

Editor.

- Templates and instantiations
- Global and local declarations
- System definition

Simulator.

- Viewers: automata animator and message sequence chart
- Control (eg, trace management)
- Variable view: shows values of the integer variables and the clock constraints defining symbolic states

Verifier.

- (see next session)

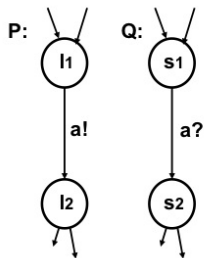
Extensions (modelling view)

- **templates** with **parameters** and an **instantiation mechanism**
- **data expressions** over **bounded integer variables** (eg, `int[2..45] x`) allowed in **guards**, **assignments** and **invariants**
- rich set of **operators** over integer and booleans, including bitwise operations, arrays, initializers ... in general a whole **subset of C** is available
- **non-standard** types of **synchronization**
- **non-standard** types of **locations**

Extension: broadcast synchronization

- A sender can synchronize with an arbitrary number of receivers
- Any receiver that can synchronize in the current state must do so
- Broadcast sending is never blocking (the send action can occur even with no receivers).

Extension: urgent synchronization

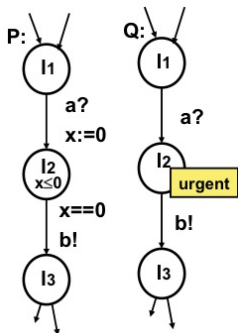


Channel a is declared **urgent chan a** if both edges are to be taken as soon as they are ready (**simultaneously** in locations l_1 and s_1).

Note the problem can **not** be solved with **invariants** because locations l_1 and s_1 can be reached at different moments

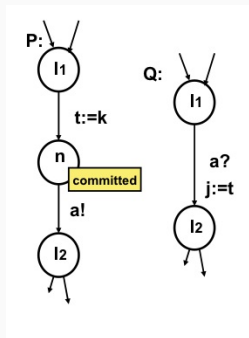
- No delay allowed if a synchronization transition on an urgent channel is enabled
- Edges using urgent channels for synchronization cannot have time constraints (ie, clock guards)

Extension: urgent location



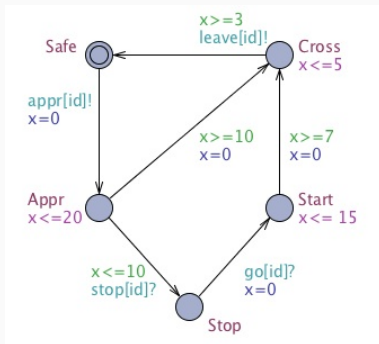
- Time does not progress but interleaving with normal location is allowed
- Both models are equivalent: **no delay at an urgent location**
- but the use of **urgent location** reduces the number of clocks in a model and simplifies analysis

Extension: committed location



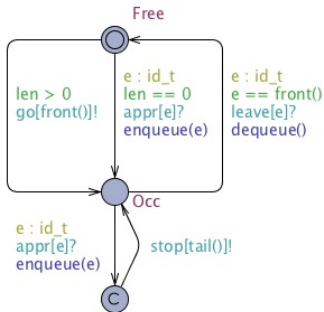
- delay is not allowed and the committed transition must be left in the next instant (or one of them if there are several), i.e., next transition must involve an outgoing edge of at least one of the committed locations
- Our aim is to pass the value k to variable j (via global variable t)
- Location n is **committed** to ensure that no other automata can assign j before the assignment $j := t$

The train gate example



- Events model approach/leave, order to stop/go
- A train can not be stopped or restart instantly
- After approaching it has 10m to receive a stop.
- After that it takes further 10 time units to reach the bridge
- After restarting takes 7 to 15m to reach the cross and 3-5 to cross

The train gate example



- Note the use of parameters and the select clause on transitions
- Programming ...

Time-critical reactive systems (Verification)

José Proença

MAPi 2019/20

October 2019

HASLab/INESC-TEC & CISTER/ISEP, Portugal

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

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

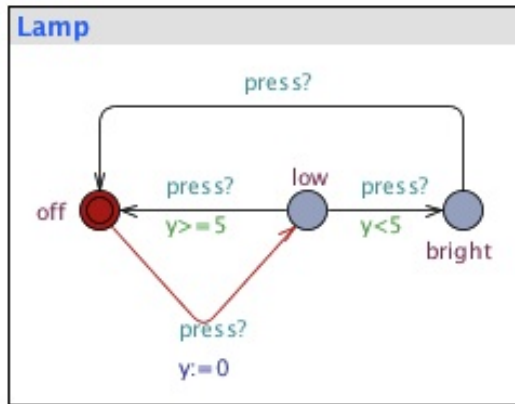
with $t_0 = 0$ and, for all clock x , $\eta_0 x = 0$.

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 .

Write possible traces

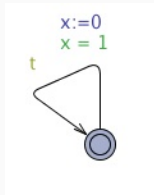
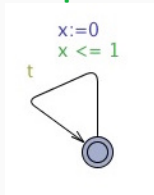


Given a **timed trace** tc , the corresponding **untimed trace** is $(\pi_2)^\omega tc$.

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



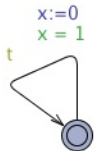
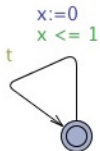
are not **timed-language equivalent**

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.

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 ,

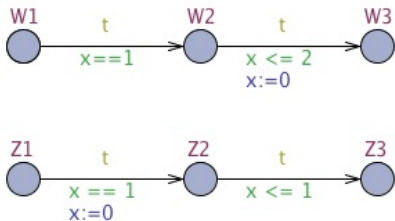
$$s_1 \xrightarrow{a} s'_1 \Rightarrow \text{there is a transition } s_2 \xrightarrow{a} s'_2 \wedge s'_1 R s'_2$$

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

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

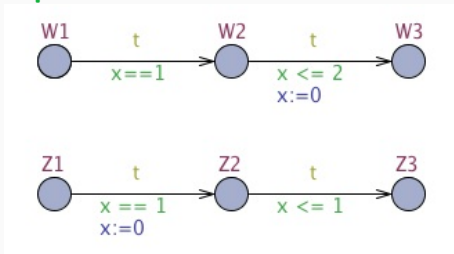
Bisimulation

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

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

Untimed bisimulation

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

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

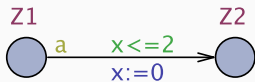
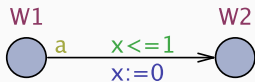
$$s_1 \xrightarrow{d} s'_1 \Rightarrow \text{there is a transition } s_2 \xrightarrow{d'} s'_2 \wedge s'_1 R s'_2$$

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 .

Untimed Bisimulation

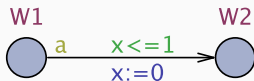
Example



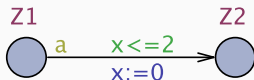
W1 bisimilar to Z1?

Untimed Bisimulation

Example



W1 bisimilar to Z1?



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

where

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

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

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

Path formulae

$$\Pi ::= A \square \Psi \mid A \diamond \Psi \mid E \square \Psi \mid E \diamond \Psi \mid \Phi \rightsquigarrow \Psi$$

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

where

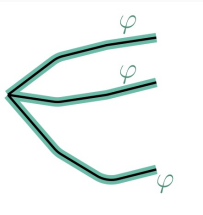
- A, E quantify (universally and existentially, resp.) over **paths**
- \square, \diamond quantify (universally and existentially, resp.) over **states in a path**

also notice that

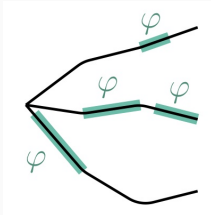
$$\Phi \rightsquigarrow \Psi \stackrel{\text{abv}}{=} A \square (\Phi \Rightarrow A \diamond \Psi)$$

Expressing properties: Uppaal

$A \square \varphi$



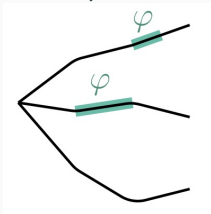
$A \diamond \varphi$



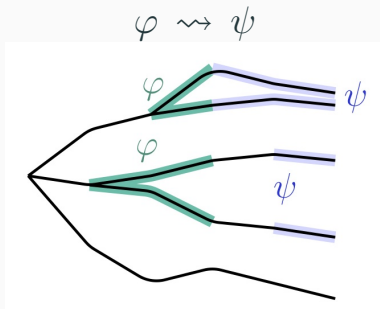
$E \square \varphi$



$E \diamond \varphi$



Expressing properties: Uppaal



Example

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

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 lose.

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

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

Something good will *eventually happen*

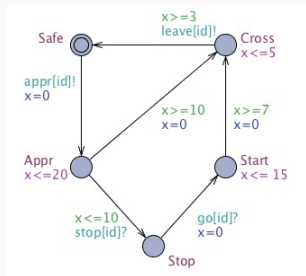
or if *something* happens, then *something else* will eventually happen!

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.

Examples: proving mutual exclusion

The train gate example

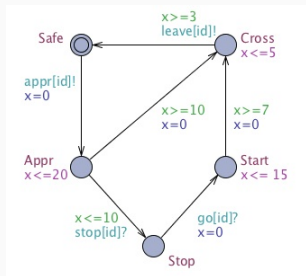


(Train 0 can reach the cross)

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

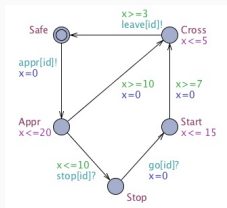
(Train 0 can cross bridge while the other trains are waiting to cross)

The train gate example



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

The train gate example



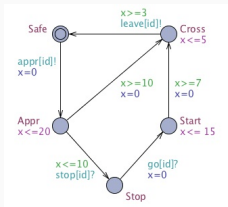
There can never be N elements in the queue

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



- `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

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](#)

Fisher's algorithm

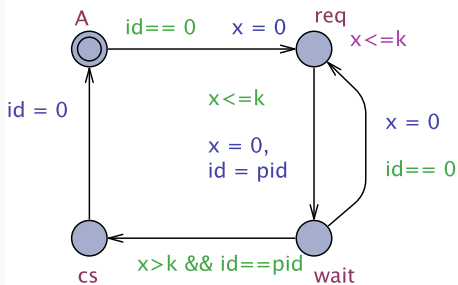
The algorithm

```
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 .

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

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**.

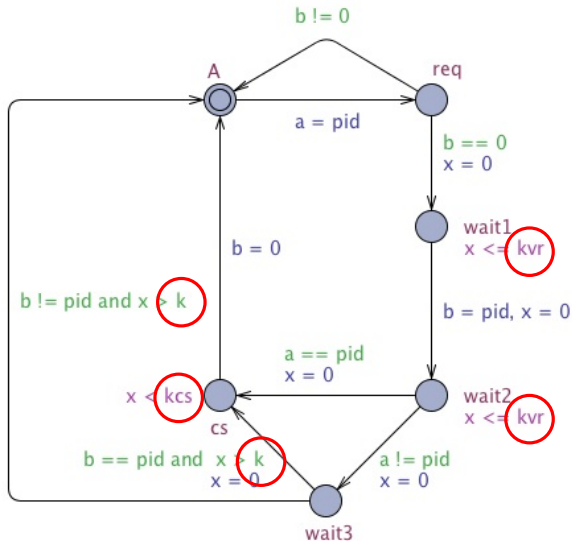
The algorithm

```
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'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

$$k \geq kvr + kcs$$

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