

# Time-critical reactive systems

Luís Soares Barbosa and **Guillermina Cledou** and Renato  
Neves and José Proença



Universidade do Minho



**Architecture and Calculi Course Unit**

# Recall

- Timed automata: syntax
  - $\langle L, L_0, Act, C, Tr, Inv \rangle$
  - $g ::= x \square n \mid x - y \square n \mid g \wedge g \mid true$
- Composition  $ta_1 \parallel_H ta_2$
- Timed automata: semantics
  - Timed Labelled Transition Systems
  - $\mathcal{T}(ta)$  with states  $\langle \text{location}, \text{clock valuation} \rangle$
- Timelocks and zeno paths
- Equivalences
  - timed traces
  - (un)timed bisimulation
- Today: UPPAAL + verification

## Note

- 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  $ta$
- ... *i.e.* 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  $ta$
- ... *i.e.* 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  $\rho$  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  $\rho$  a path and  $\delta$  a label in  $\mathcal{T}(ta)$

# Timelock paths

## Definition

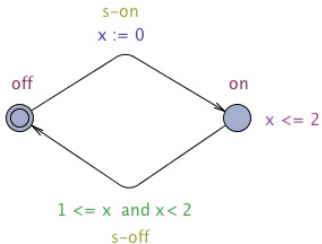
A path is **timelock** if it contains a state with a timelock, ie, a **state from which there is not any time-divergent path**

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

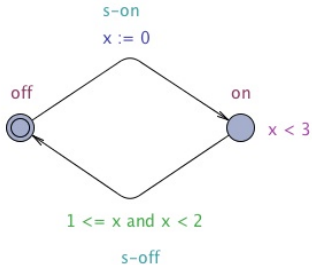


State  $\langle on, 2 \rangle$  is reachable through path

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

and is terminal

# Timelock paths



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

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

# Zeno

## 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)

# Zeno

## 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

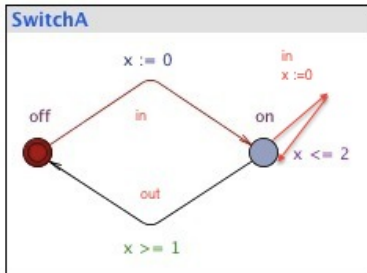
An infinite path fragment  $\rho$  is **zeno** if it is **time-convergent** and **infinitely many actions occur along it**

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

# Zeno

## 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 ...)

# Zeno

## 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:

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

# Warning

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.

# Time-critical reactive systems (Timed-automata in UPPAAL)

Luís Soares Barbosa and **Guillermina Cledou** and Renato  
Neves and José Proença



Universidade do Minho



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

**Architecture and Calculi Course Unit**



# 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

# Extensions: Data Declarations

## Data Variables (with bounded domains)

```
int x,y;  
bool b = false;  
int[0,5] n,arr1[2];  
int arr2[2];
```

## Constants

```
const int N = 5;
```

## Type Definitions

```
typedef int[0,4] user_id_t;
```

## Extension: Channels

Channels can be seen as enriched actions of a TA

```
chan press;
```

Channel `press` encodes two complementary actions:

- `press!` (sender/output)
- `press?` (receiver/input)

## Extension: Different kinds of synchronization

- **handshake** (**standard**): two transitions with complementary actions

```
chan press;
```

- **broadcast**: a transition with a sender and any number of transitions with its complementary receiver

```
broadcast chan press;
```

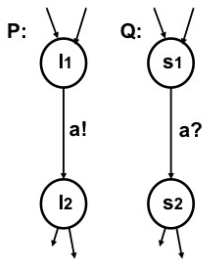
- **urgent**: similar to handshake but without delaying

```
urgent chan press;
```

## 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

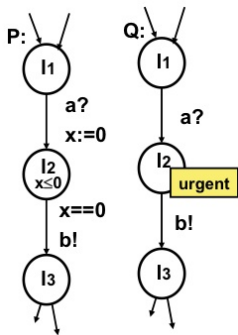


Channel  $a$  is declared **urgent chan  $a$**  if both edges are to be taken as soon as they are ready (**simultaneously** in locations  $l_1$  and  $s_1$ ).

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)

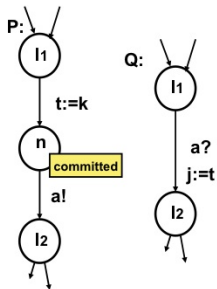
## Extension: urgent location



- 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

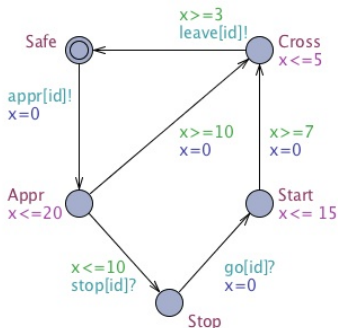


## Extension: committed location



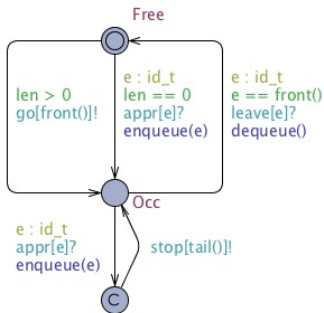
- delay is not allowed and the committed transition must be left in the next instant (or one of them if there are several), i.e., next transition must involve an outgoing edge of at least one of the committed locations
- Our aim is to pass the value  $k$  to variable  $j$  (via global variable  $t$ )
- 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 ...

# Excercise

## Coffee Machine

- it can serve **coffee** or **cappuchino**
- if a **coin** is inserted, it waits 5 seg. for the user to choose or it **cancel**s the request
- it takes between 2 and 5 seg. to prepare a drink, after which it **serve**s it

# Time-critical reactive systems (Verification)

Luís Soares Barbosa and **Guillermina Cledou** and Renato  
Neves and José Proença



Universidade do Minho



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

**Architecture and Calculi Course Unit**

# Properties: expression and satisfaction

## The satisfaction problem

Given a **timed automata**,  $ta$ , and a **property**,  $\phi$ , show that

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

# Properties: expression and satisfaction

## The satisfaction problem

Given a **timed automata**,  $ta$ , and a **property**,  $\phi$ , show that

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

- in which logic language shall  $\phi$  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):

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

Additionally,

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

# Expressing properties: UPPAAL

## Path formulae

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

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

where

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

also notice that

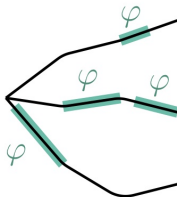
$$\Phi \rightsquigarrow \Psi \stackrel{\text{abv}}{=} A\Box(\Phi \Rightarrow A\Diamond\Psi)$$

# Expressing properties: UPPAAL

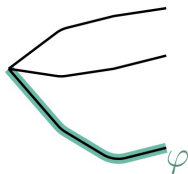
$A\Box\varphi$



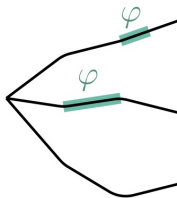
$A\Diamond\varphi$



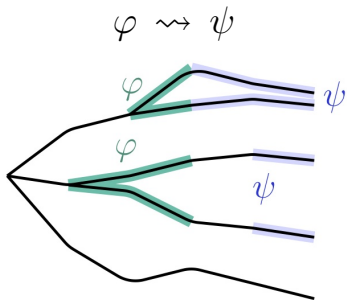
$E\Box\varphi$



$E\Diamond\varphi$



# 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  $\phi$  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.

# Liveness properties

$$A \diamond \phi \text{ and } \phi \rightsquigarrow \psi$$

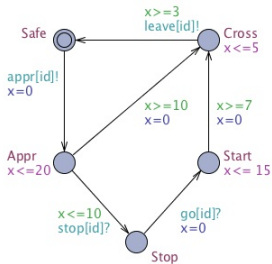
Something good will *eventually happen*

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.

# The train gate example



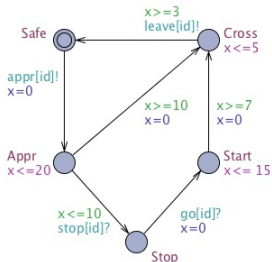
(Train 0 can reach the cross)

(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)

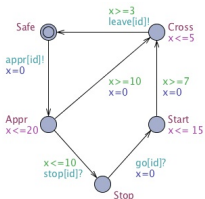


# The train gate example



- $E \langle \rangle \text{Train}(0).\text{Cross}$   
(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



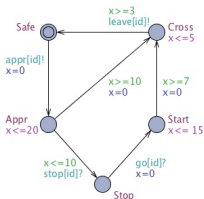
There can never be N elements in the queue

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

# 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.

# 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:

- [Fischer's protocol](#) (included in the UPPAAL distribution)
- [Lamport's protocol](#)

# Fischer's algorithm

## The algorithm

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

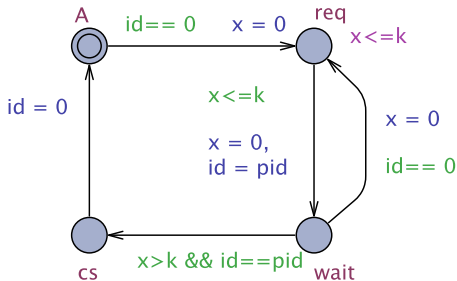
# Fischer's algorithm

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



## Fischer's algorithm in UPPAAL



- 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

# Fischer'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

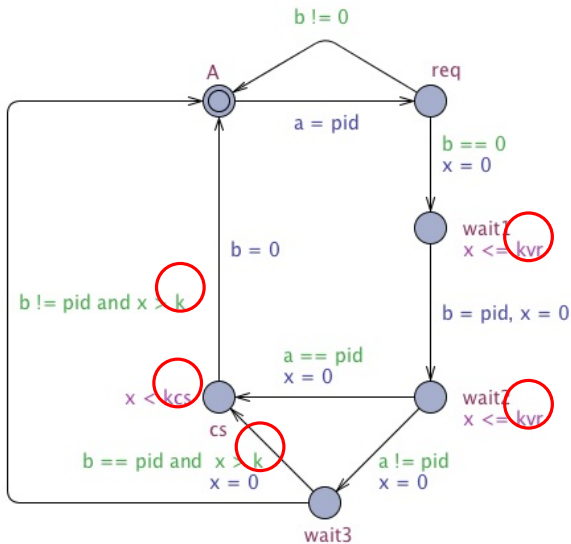
```
start :  $a := i$   
       if  $b \neq 0$  then goto start  
        $b := i$   
       if  $a \neq i$  then delay( $k$ )  
           if  $b \neq i$  then goto start  
       (critical section)  
        $b := 0$ 
```

# 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

$$k \geq kvr + kcs$$

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