4. Verification of Real-Time systems in UPPAAL

Renato Neves   José Proença

CPC 2022/2023

Cyber Physical Computation

CISTER – ISEP, Porto, Portugal

U.Minho, Braga, Portugal

https://lmf.di.uminho.pt/CyPhyComp2223/

https://haslab.github.io/MFP/PCF/2223/

Syllabus

- CSS: a simple language for concurrency
  - Syntax
  - Semantics
  - Equivalence
- Timed Automata
  - Syntax
  - Semantics (composition, Zeno)
  - Equivalence
  - UPPAAL tool
    - Specification
    - CTL and Verification
- A simple C-like language
  - Syntax
  - Semantics (operational)
- Hybrid-language: adding differential equations
  - Syntax
  - Semantics
  - Lince tool
    - Specification
    - Analysis
- Monads: semantics with computational effects

Table of contents

1. Modelling in UPPAAL
2. Behavioural Properties
3. Examples: proving mutual exclusion

Modelling in Uppaal

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)`

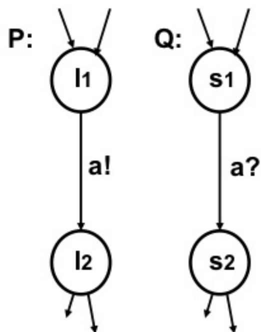
Extensions (modelling view)

- templates with parameters and an instantiation mechanism
- data expressions over bounded integer variables (eg, `int[2..45] x`) allowed in guards, assigments 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 than can synchronize in the current state must do so
- Broadcast sending is never blocking (the send action can occur even with no receivers).

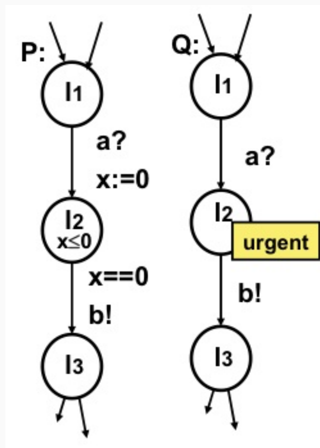
Extension: urgent synchronization



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

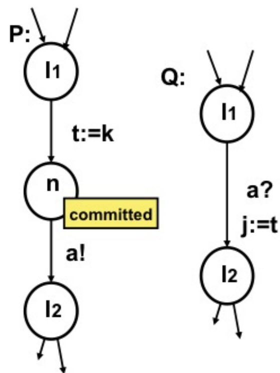
- No delay allowed if a synchronization transition on an urgent channel is enabled
- Edges using urgent channels for synchronization cannot have time constraints (ie, clock guards)

Extension: urgent location



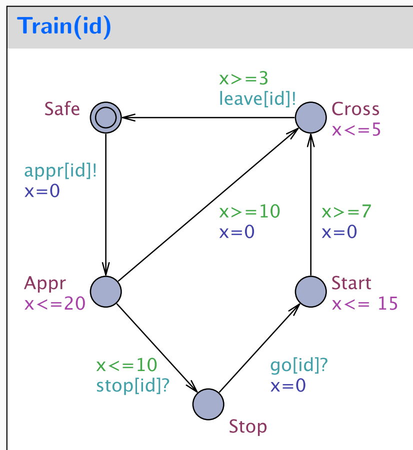
- Time does not progress but interleaving with normal location is allowed
- Both models are equivalent: no delay at an urgent location
- but the use of urgent location reduces the number of clocks in a model and simplifies analysis

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 $j := t$

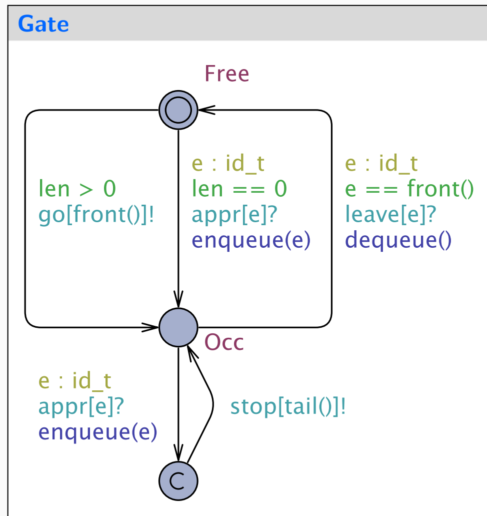
The train-gate example



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

The train-gate example

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



Behavioural Properties

Properties: expression and satisfaction

The satisfaction problem

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

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

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

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

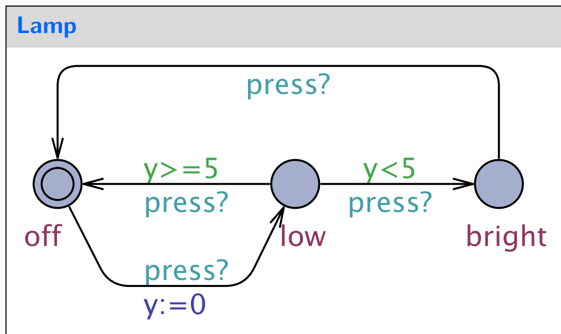
Any expression which can be evaluated to a boolean value for a state (typically involving the clock constraints used for guards and invariants and similar constraints over integer variables):

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

Additionally,

- $ta.\ell$ 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 $\langle \ell, \eta + d \rangle$

Exercises



Ex. 4.1: Write a state formula

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

Expressing properties: Uppaal

Path formulae

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

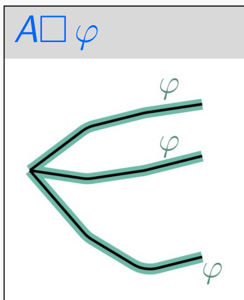
where

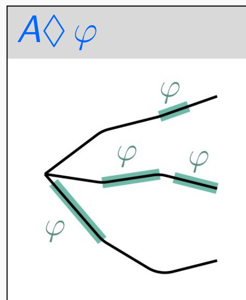
- A, E quantify (universally and existentially, resp.) over paths
- $\Box$, $\Diamond$ quantify (universally and existentially, resp.) over states in a path

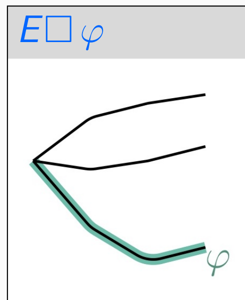
also notice that

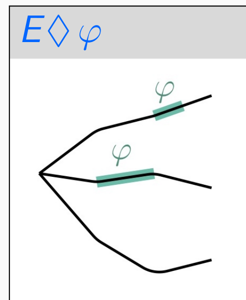
$$\Phi \leadsto \Psi \;\stackrel{\text{abv}}{=}\; A\Box\,(\Phi \Rightarrow A\Diamond\,\Psi)$$

Expressing properties: Uppaal

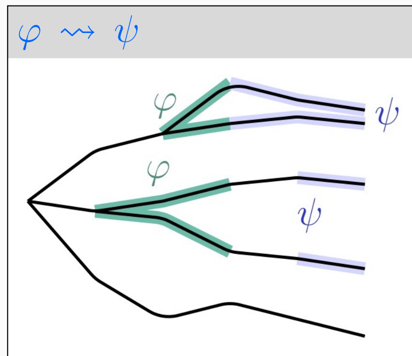








Expressing properties: Uppaal



Example

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

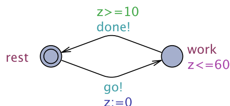
$A\Diamond\,\phi$ and $\phi \leadsto \psi$

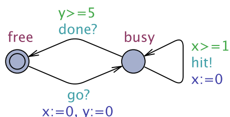
Something good will *eventually happen*
or if *something* happens, then *something else* will eventually happen!

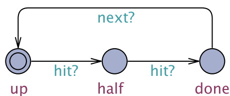
Examples

- 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.

Exercise: worker, hammer, nail - revisited





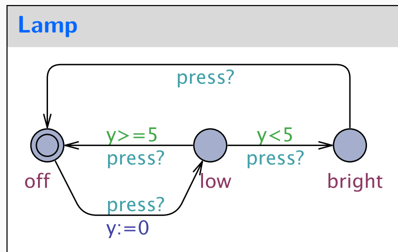


Ex. 4.2: Write properties and explain them

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

(Practice in UPPAAL)

Exercise: write formulas

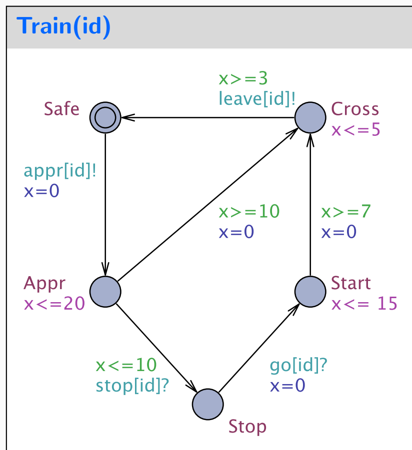


Ex. 4.3: Write formulas, and say which ones are true

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

The train gate example (1/2)

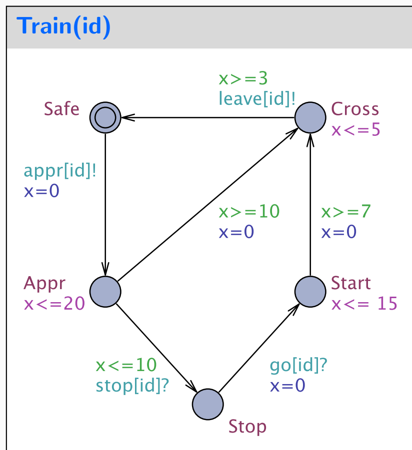


(Train 0 can reach the cross)

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

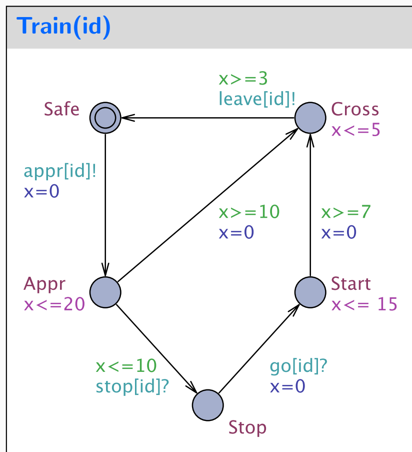
(Train 0 can cross bridge while the other trains are waiting to cross)

The train gate example (1/2)



- `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 (2/2)



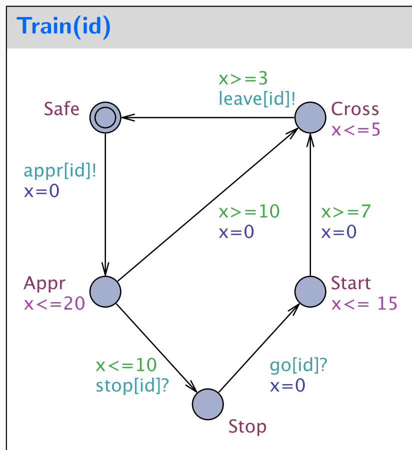
There can never be N elements in the queue

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[] 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:

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

Fisher's algorithm

The algorithm

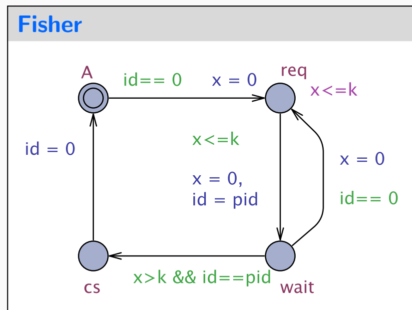
```
repeat
    repeat
        await id = 0
        id := i
        delay(k)
    until id = i
    (critical section)
    id := 0
forever
```

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

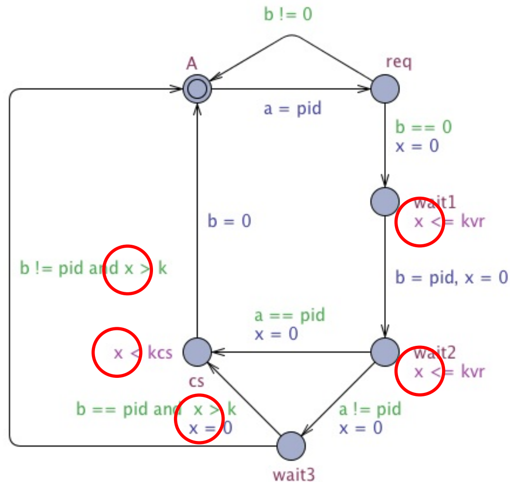
$$
\begin{aligned}
\text{start}: \ & a := i \\
& \text{if } b \neq 0 \text{ then goto start} \\
& b := i \\
& \text{if } a \neq i \text{ then delay}(k) \\
& \qquad\qquad \text{else if } b \neq i \text{ then goto start} \\
& \textit{(critical section)} \\
& b := 0
\end{aligned}
$$

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