Introduction to Lab 4

Modelling and Verification using UPPAAL

Syed Md Jakaria Abdullah <jakaria.abdullah@it.uu.se>
Gaoyang Dai <gaoyang.dai@it.uu.se>

9. October 2018

Lab 4: Modelling and Verification using UPPAAL

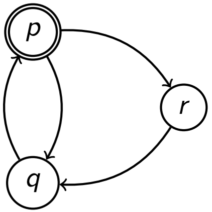
- Lab goals:
  - Practice formal modelling and verification of RTS
  - Work with timed automata and UPPAAL
- Lab preparation:
  - Work in your groups
  - Lab will be done on Friday, Oct 12, from 10:15 to 12:00. in room 1515, ITC.
  - Have a look at the lab homepage `http://www.it.uu.se/edu/course/homepage/realtid/2017-368/labs/lab4`
  - (UPPAAL tutorial on the page is recommended reading!)
- Lab report:
  - Answers (models, queries, values) to the questions.
  - A report should be hand in through student portal.
  - *Deadline: Friday, Oct 19, 23:59*

Lab Assignment

- Part 1: Warm-Up
  - Model 3 simple automata
  - Use verification for simple properties
- Part 2: Scheduling
  - Setting: Schedule jobs to CPUs
  - One automaton *per job* and *per CPU*
  - Determine minimal execution time
- Part 3: Deadlock detection
  - Model Buffer, Producer and Consumer from Ada lab
  - Use verifier to *find deadlocks*
    - ⋆ "Deadlock" means: Only time may pass (for all future)
  - Use simulator to analyze them
  - Remove all deadlocks

Finite Automata

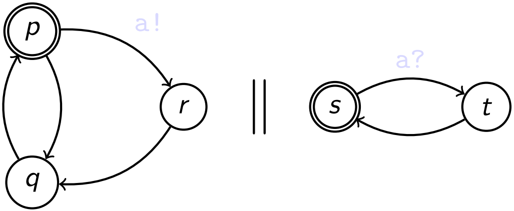
- Theoretic model for systems (or whatever else)
- *Locations* and *transitions* (drawn as nodes and edges)



- State space: Set of locations
- Trace semantics:
  - One possible trace: $p \to q \to p \to r \to q \to \ldots$
  - Another one: $p \to r \to q \to p \to r \to \ldots$
  - *Not* a trace: $p \to r \to p \to \ldots$

Networks of Finite Automata

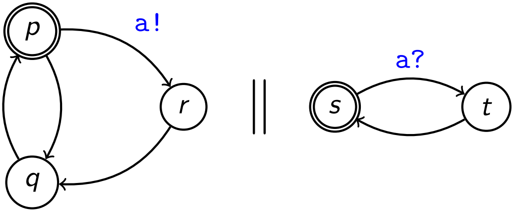
- Compose several automata into *networks*
- Use *synchronization* on edges/transitions



- State space: Product of location sets
- Trace semantics:
  - Interleaving, i.e., one automaton at a time
  - Except: Synchronized edges are taken *together*
  - E.g.: $(p, s) \rightarrow (q, s) \rightarrow (p, s) \xrightarrow{a} (r, t) \rightarrow (r, s) \rightarrow \ldots$
  - *Not* a trace: $(p, s) \rightarrow (r, s) \rightarrow \ldots$

Networks of Finite Automata

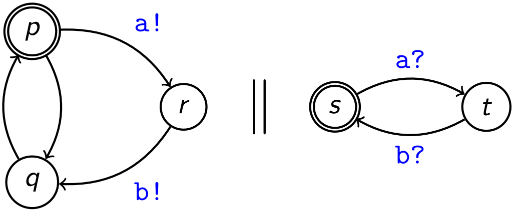
- Compose several automata into *networks*
- Use *synchronization* on edges/transitions



- State space: Product of location sets
- Trace semantics:
  - Interleaving, i.e., one automaton at a time
  - Except: Synchronized edges are taken *together*
  - E.g.: $(p, s) \to (q, s) \to (p, s) \xrightarrow{a} (r, t) \to (r, s) \to \dots$
  - *Not* a trace: $(p, s) \to (r, s) \to \dots$

Finite Automata: Model Checking

- Does a model *satisfy* some property φ?



Automaton A $\quad\|\quad$ Automaton B

- Property: "Does $A.r$ imply $B.t$?"
  - $\varphi :=$ `A[] (A.r imply B.t)`
  - Means: "In each state of each trace, B is in t whenever A in r"
- Is *satisfied* in above example
- (Not satisfied without b synchronization!)

Temporal Logic (CTL, Computation Tree Logic)

- Temporal operators

`A[] p`: *p* is an invariant

  ★ In *all* executions, *p* *always* holds

`E[] p`: *p* may hold globally

  ★ *There is* an execution in which *p* *always* holds

`E<>` `p`: *p* is reachable/possible

  ★ *There is* an execution in which *p* *eventually* holds

`A<>` `p`: *p* is guaranteed

  ★ In *all* executions, *p* *eventually* holds

- (UPPAAL cannot nest them)

Operator = Path quantifier + State operator

- `A`, `E`: Path quantifiers (**A**lways, **E**ventually)
- `[]`, `<>`: State operators (often written `G`, `F`: **G**lobally, **F**inally)

Temporal Logic (CTL, Computation Tree Logic)

- Temporal operators

`A[] p`: *p* is an invariant

  - ⋆ In *all* executions, *p* *always* holds

`E[] p`: *p* may hold globally

  - ⋆ *There is* an execution in which *p* *always* holds

`E<> p`: *p* is reachable/possible

  - ⋆ *There is* an execution in which *p* *eventually* holds

`A<> p`: *p* is guaranteed

  - ⋆ In *all* executions, *p* *eventually* holds

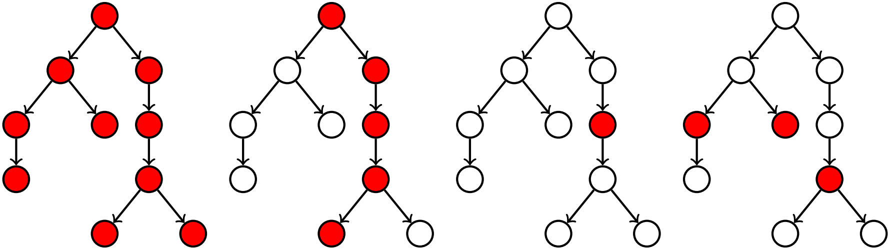
- (UPPAAL cannot nest them)

Operator = Path quantifier + State operator

- `A`, `E`: Path quantifiers (**A**lways, **E**ventually)
- `[]`, `<>`: State operators (often written `G`, `F`: **G**lobally, **F**inally)

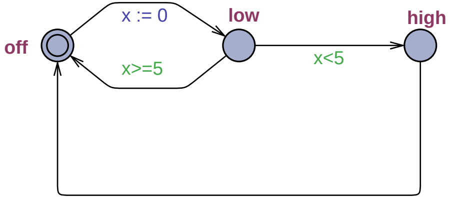
Temporal Operators

`A[] p` `E[] p` `E<> p` `A<> p`



Timed Automata

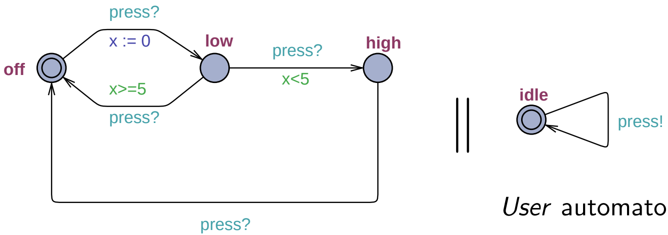
- Extend finite automata with *clocks*:



- Clocks have *real* values
  - All increasing at same pace
  - Can be reset and compared
- State space: Location $\times$ Clock valuations
- Trace semantics: Additional *delay* transitions
  - $(off, 0) \xrightarrow{\delta} (off, 1.2) \rightarrow (low, 0.0) \xrightarrow{\delta} (low, 5.7) \rightarrow (off, 5.7) \rightarrow (low, 0.0) \xrightarrow{\delta} (low, 2.3) \rightarrow (high, 2.3) \rightarrow \dots$

Networks of Timed Automata

- Compose just like before, using synchronized edges



Lamp automaton

User automaton

- In UPPAAL:
  - Sync. channels need to be *declared*
  - (As well as clocks and variables)

UPPAAL

- *Model Checker* for timed automata
  - Developed at Aalborg University, Denmark and Uppsala University
  - Started 1995, rather mature by now
  - Different branches: Timed games, costs, statistical model checker, ...
  - GUI in Java, verification engine C++
  - *Extensive online help.* Use it!
- Three panes:
  1. Automata editor
  2. Simulator
  3. Verifier
- Free for private/academic use (but closed-source)
- You can run it at home: `http://www.uppaal.org`

UPPAAL (cont.)

Demo

The End

Questions?