Lecture: Modelling and Verification (Advanced Topic)

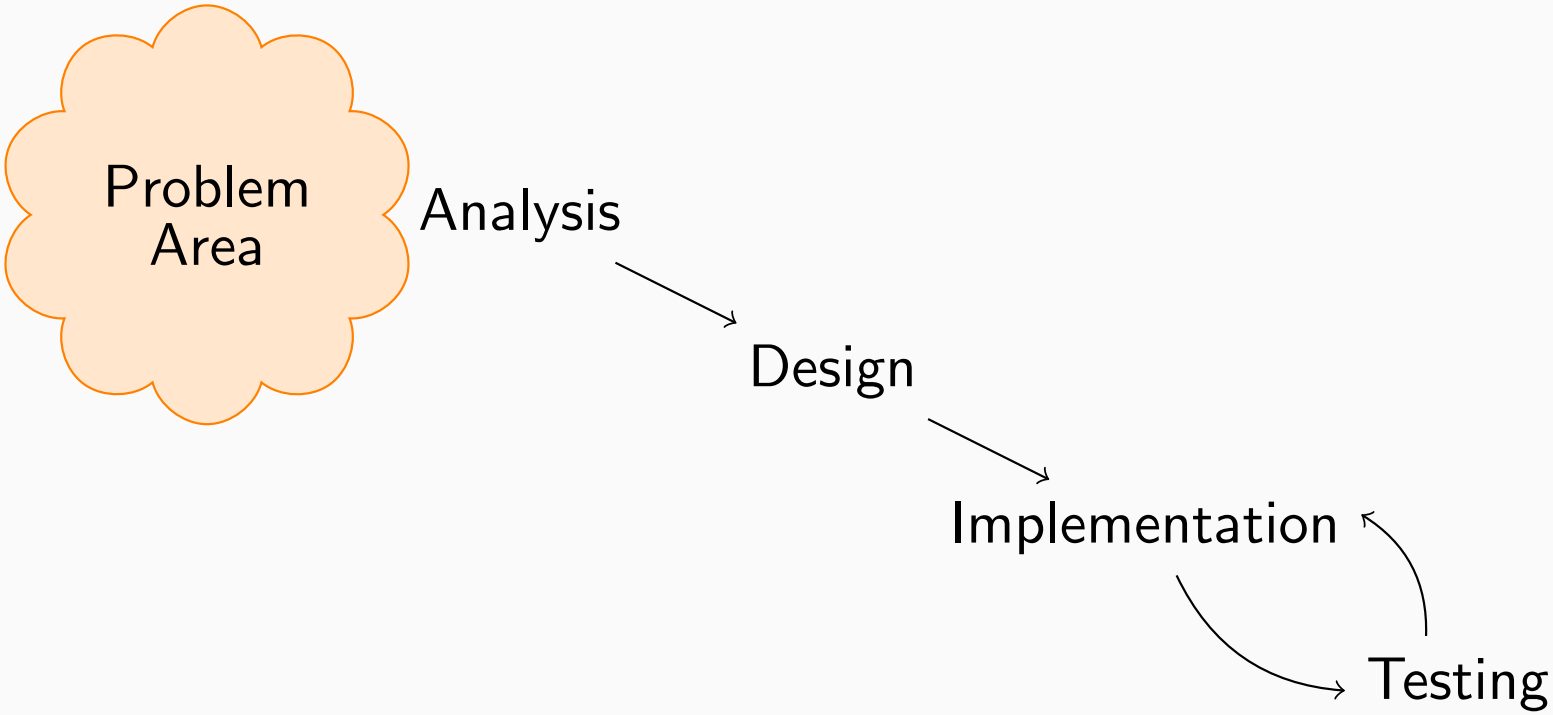
Real-Time Systems, HT15

Philipp Rümmer

Slides mainly by: Martin Stigge

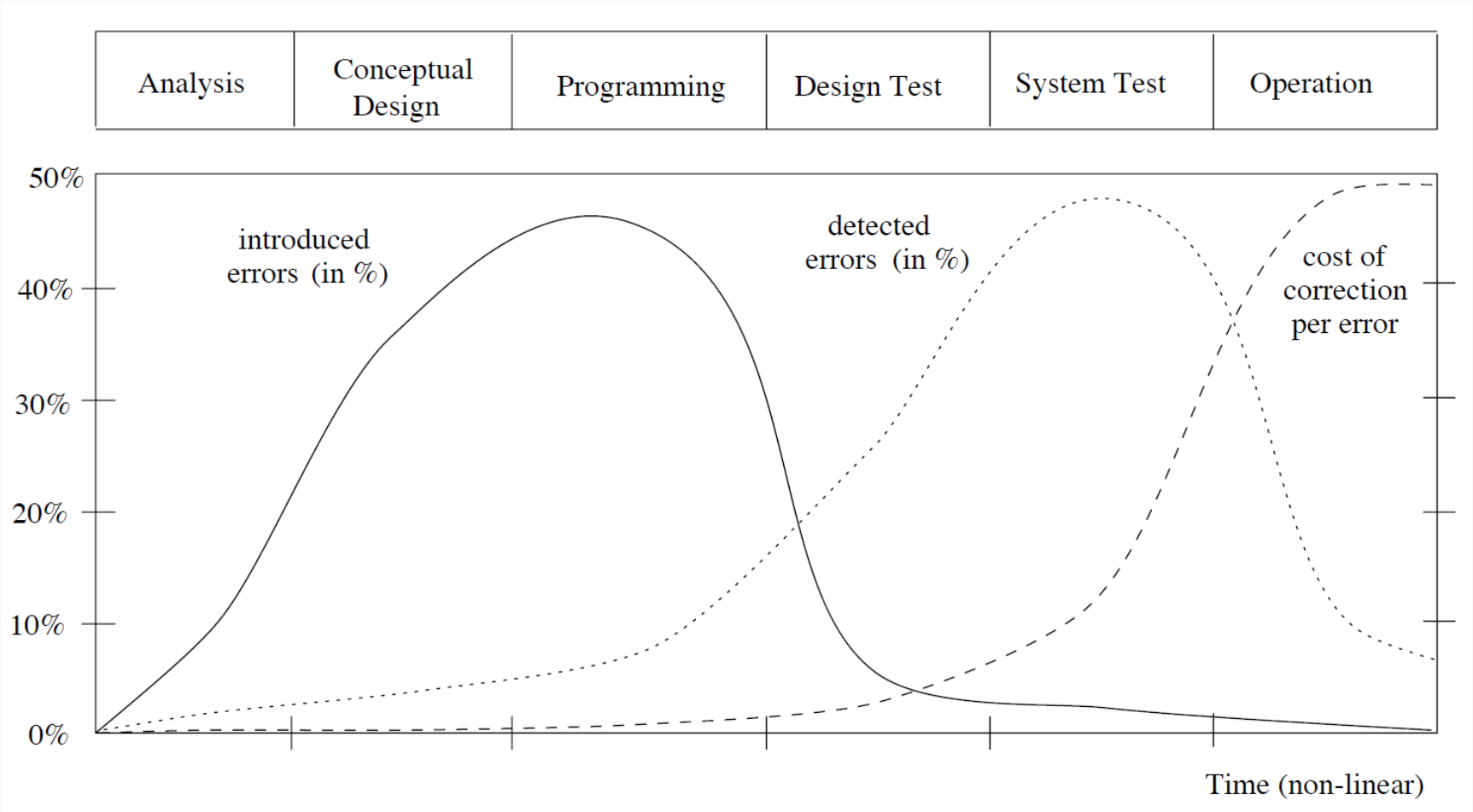
30. September 2015

Software Development Process (Traditionally..)



- Errors are detected *late*
- The later the more expensive
- 30-50% of the time: *Testing!*

Error Introduction, Detection and Correction

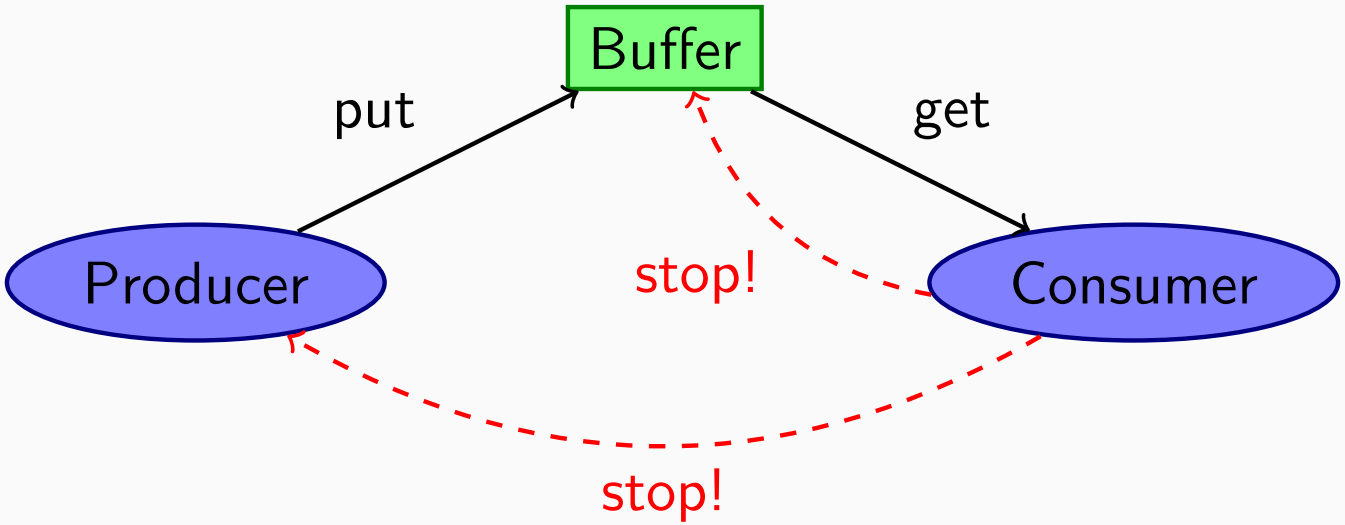


- *Detect earlier?*

Introduce Formal Modelling

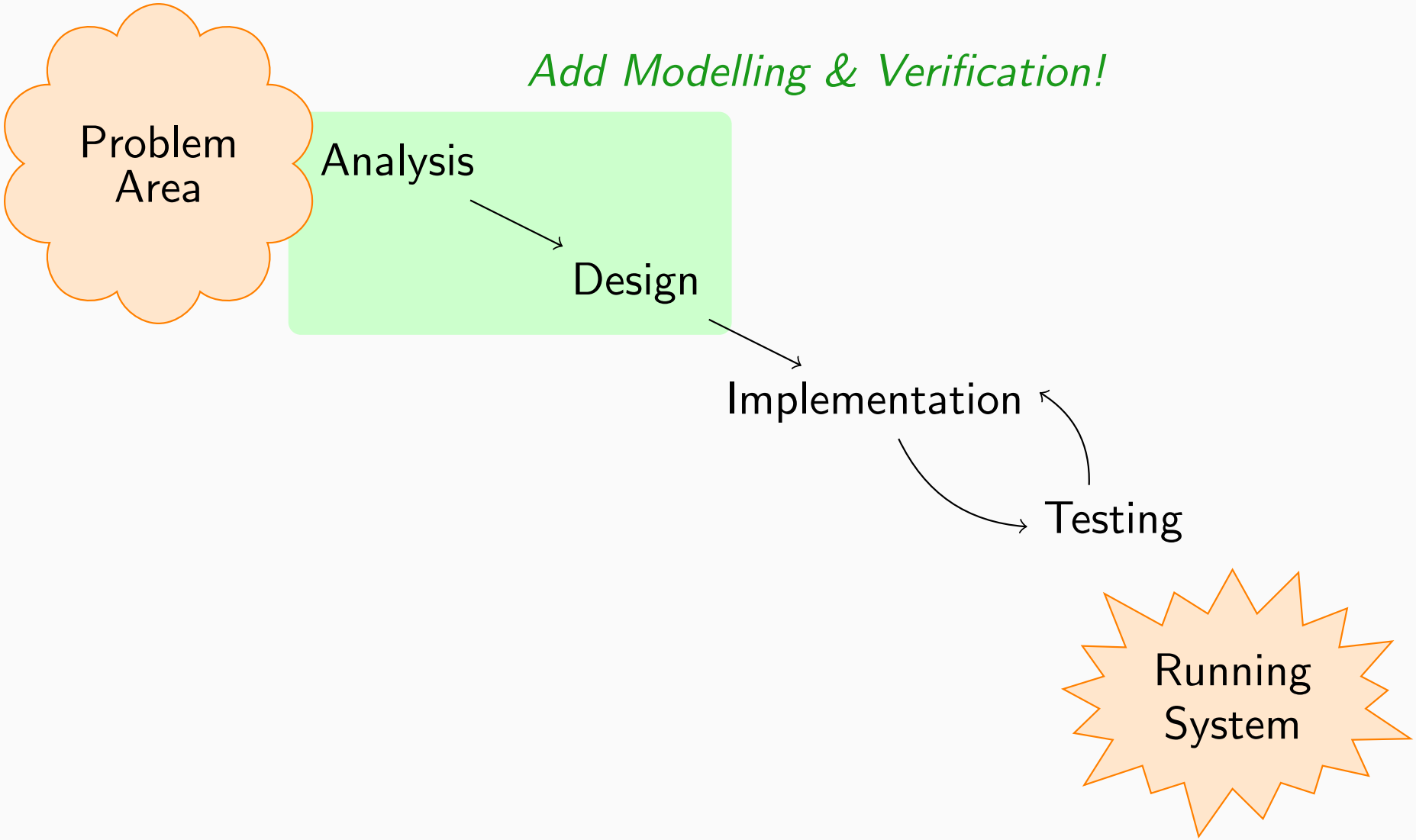
- Formal Modelling early in the design process
- Use *verification* to detect errors
  - (Potentially) reduces cost and time!
- Emerging field in Computer Science
  - Methods and tools now available

Example: Deadlock in Ada Assignment



- Consumer at sum 100: Tell everyone to stop
- ... *In which order?*
  - Buffer first
    - ★ Producer might communicate with terminated Buffer
  - Producer first
    - ★ Circular waiting possible
- All initial approaches flawed, generations of students clueless
  - Formal verification can *detect the problem*!

Software Development Process (Traditionally..)



Modelling . . .

System models

- Design Process: Systematic system description
- Abstraction of system behavior
- Only keep *essential* aspects
- Examples: State charts, *Timed automata,* Dataflow diagrams. . .

Properties

- In context of system model, express desired (formal) system properties
- Usually derived from requirements
- Examples: *Temporal logic,* Monitors/observers, . . .

. . . and Verification

Verification

- Check that system model satisfies properties
- Can be *exact,* or an under-/over-approximation
- Examples: Testing, Simulation, Theorem proving, *Model checking*
- Various tools: SMV, SPIN, *UPPAAL,* Simulink Design Verifier . . .

Validation (not considered here)

- Do the requirements/properties capture the right system?
- Are the requirements consistent?

V&V = Verification & Validation

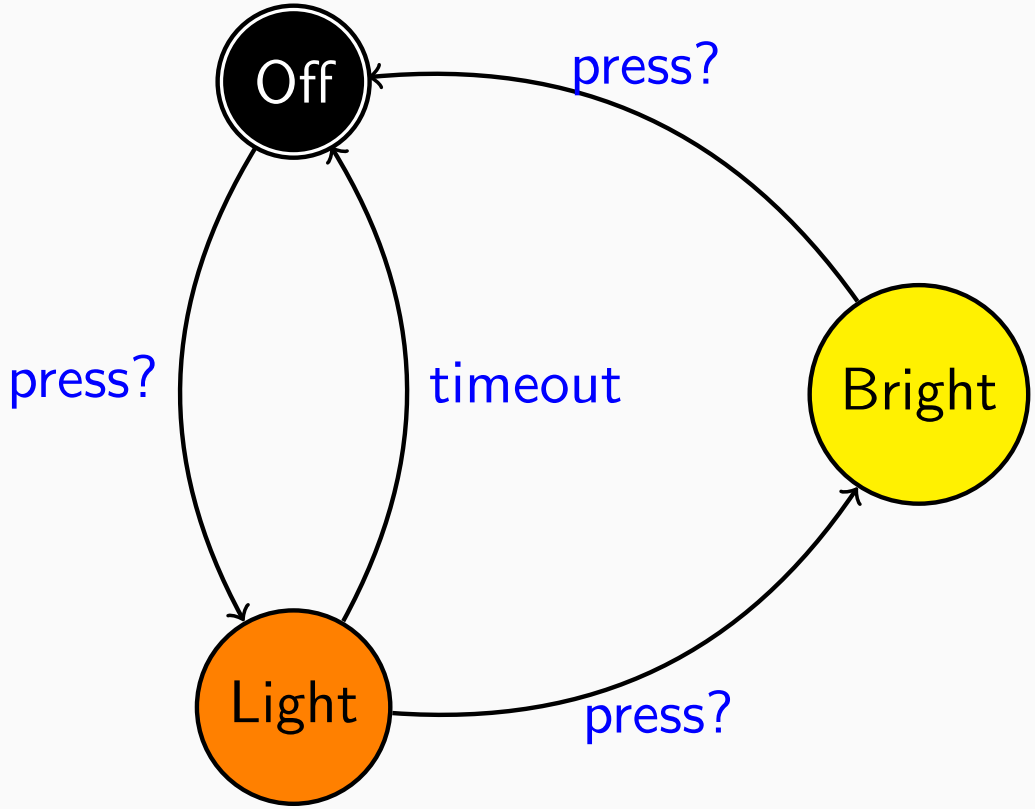
- Validation: "the right system is built"
- Verification: "the system is built right"

Fahrplan

1. Motivation
   - Errors in Software Development
   - Modelling and Verification
2. Modelling
   - Finite Automata
   - Timed Automata
   - Ada Programs as Timed Automata
3. Verification
   - Specifications
   - UPPAAL

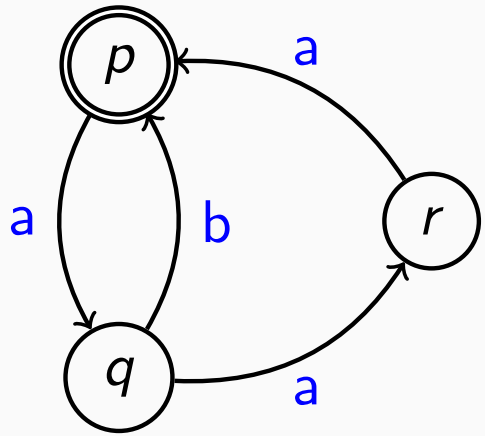
Finite Automata: First Example

- Intelligent Light Control



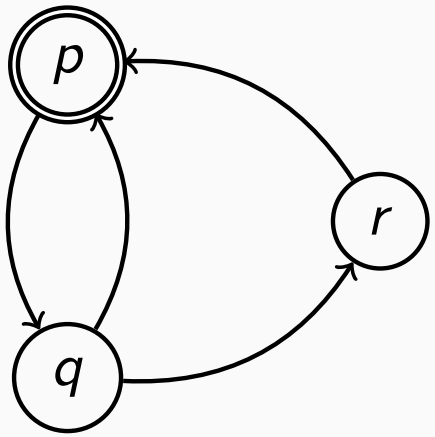
Finite Automata: More formally

- Theoretic model for systems (or whatever else)
- *Locations* and *transitions* (drawn as nodes and edges)
- *Initial* location (double circle) and *final* locations (none here)
- *Actions* (or *events*) on the transitions (drawn as edge labels)



- Accepted *language*: $(ab|aaa)*$
  - Describes sequence of actions
  - Regular language
  - Cf. your favorite book about automata theory
- View automaton as a task (actions are synchronization)

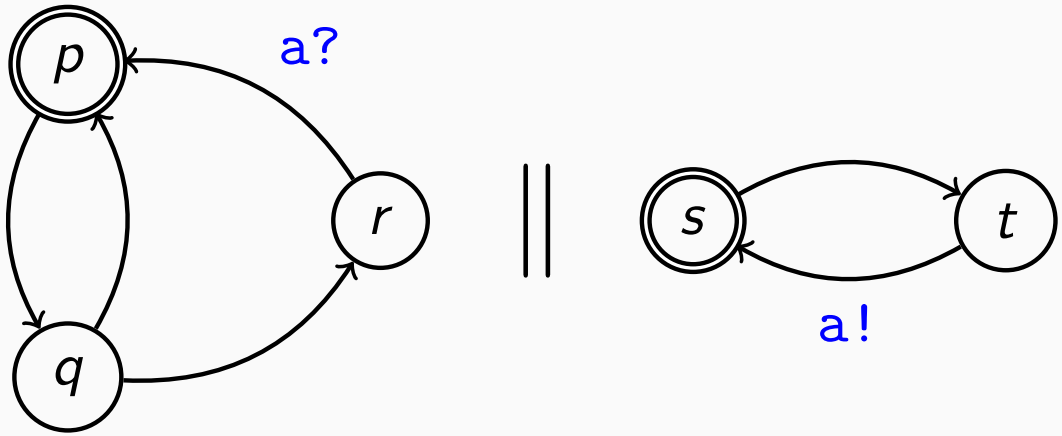
State Space



- State space: Set of locations
- Trace semantics:
  - One possible trace: $p \to q \to p \to q \to r \to p \to \ldots$
  - Another one: $p \to q \to r \to p \to q \to \ldots$
  - *Not* a trace: $p \to r \to p \to \ldots$
- Interesting if there are "good" and "bad" states

Networks of Finite Automata

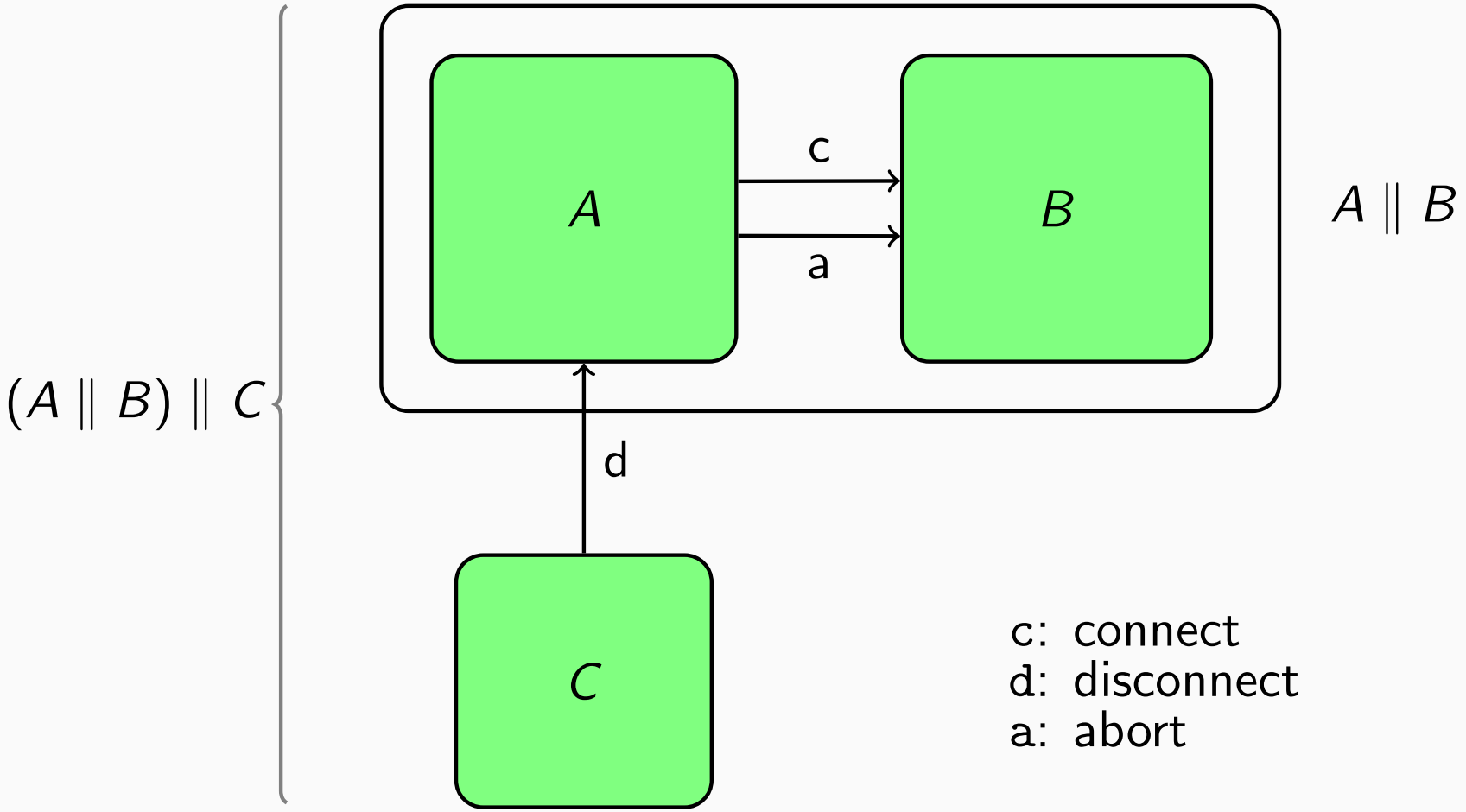
- Compose several automata into *networks*
- Use *synchronization* on edges/transitions
  - Write ? and ! for input/output actions (complementary)



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

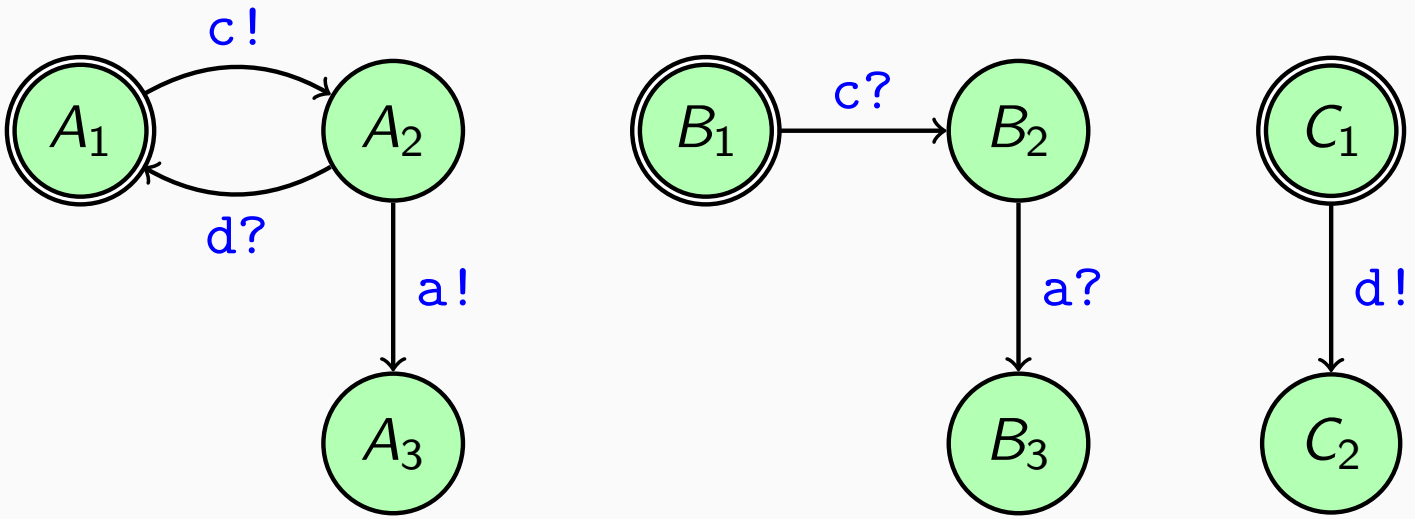
Compositional Models

- Synchronization models channels in composed models



Synchronization Example

- Sync. execution may lead to deadlocks



- Consider trace:
  $(A_1, B_1, C_1) \xrightarrow{c} (A_2, B_2, C_1) \xrightarrow{d} (A_1, B_2, C_2)$ . . . *Deadlock!*
- (Deadlock can be removed by adding another component.)

Small exercise (5min)

Vending Machine Model

Draw a finite automaton modelling the behaviour of a vending machine. The vending machine is idle until a customer inserts one or multiple coins. After that, the customer can choose a product, which is subsequently delivered by the machine. If no product is chosen within some amount of time, all coins are returned.

Add further (reasonable) details and behaviour in the model, if necessary.

State charts (e.g., UML, Simulink)

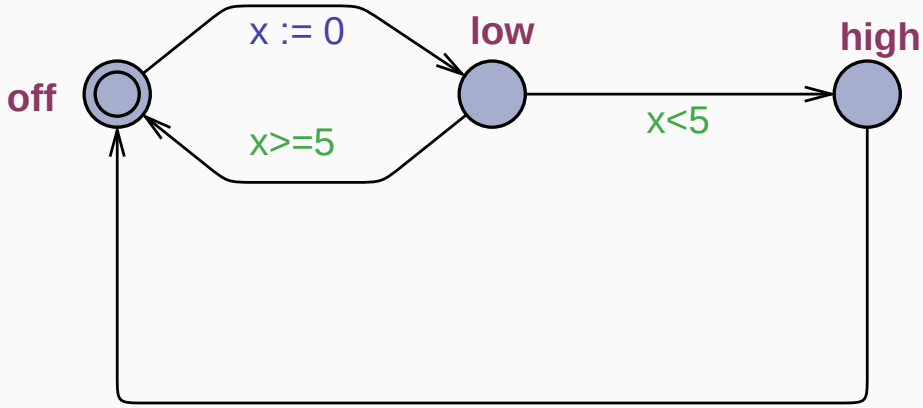
Finite automata + many further features

- Data variables
- Guards on transitions
- Function calls, state activities
- Hierarchies/nested automata
- *etc.*

- Very common modelling formalism
- Originally introduced by David Harel (1987)
- Wide range of tools for modelling and verification

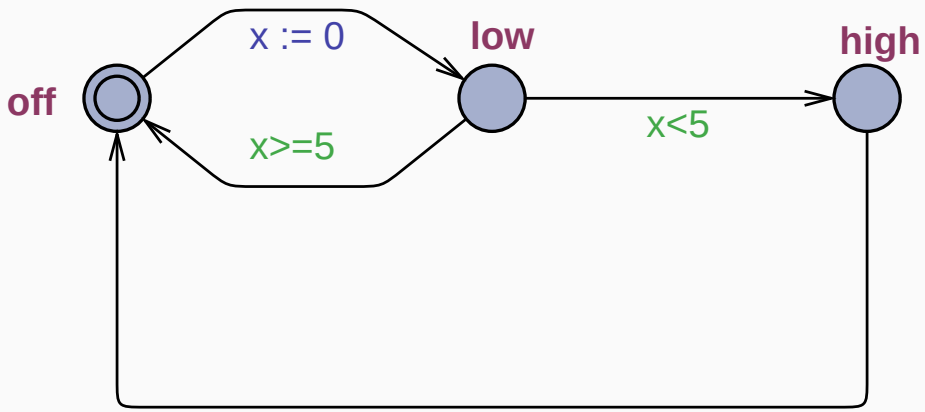
Timed Automata: Lamp Example

- Extend finite automata with *clocks*:



- A finite number of *clocks*
  - Real-valued
  - All increasing at same rate, starting from 0
  - Can be reset (`x:=0`) and compared (`x<5`)

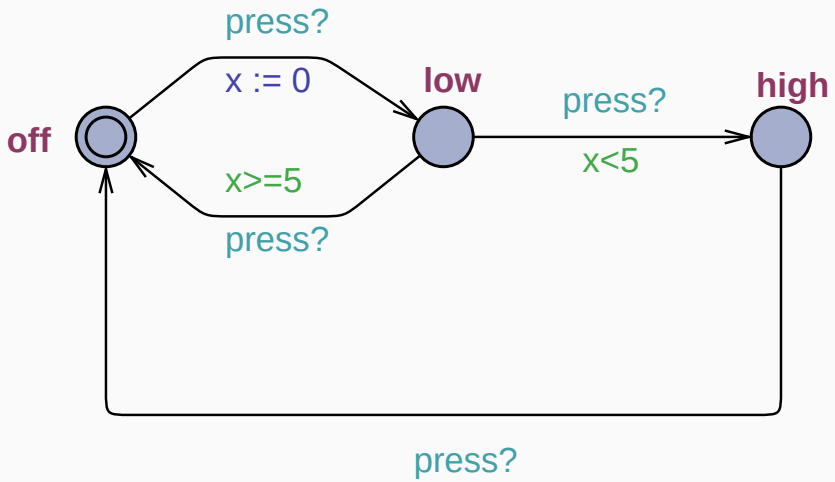
Timed Automata: Semantics



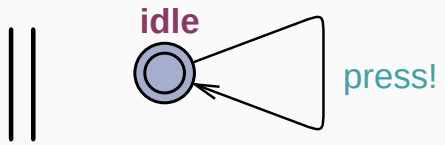
- State space: Location $\times$ Clock valuations
- Trace semantics: Additional *delay* transitions
  - $(off, 0) \xrightarrow{\delta} (off, 1.2) \rightarrow (low, 0.0) \xrightarrow{\delta} (low, 5.7) \rightarrow (off, 5.7) \rightarrow (low, 0.0) \xrightarrow{\delta} (low, 2.3) \rightarrow (high, 2.3) \rightarrow \ldots$

Networks of Timed Automata

- Compose just like before, using synchronized edges



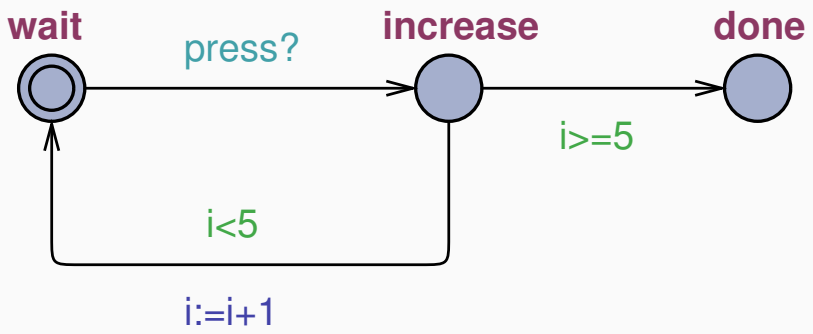
Lamp automaton



User automaton

Additional Data Variables

- Variables also possible, e.g., integers



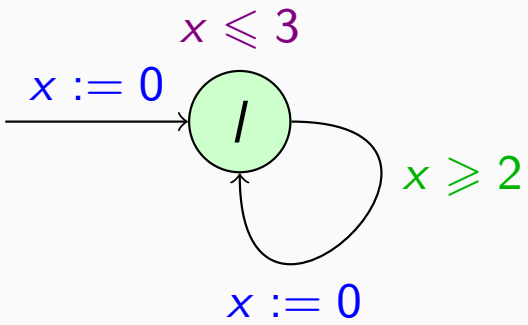
- State space: Location $\times$ Clock valuations $\times$ Data domains
- Example trace:
  - $(wait, i = 0) \rightarrow (increase, i = 0) \rightarrow (wait, i = 1) \rightarrow \ldots$
    $\rightarrow (wait, i = 5) \rightarrow (increase, i = 5) \rightarrow (done, i = 5)$

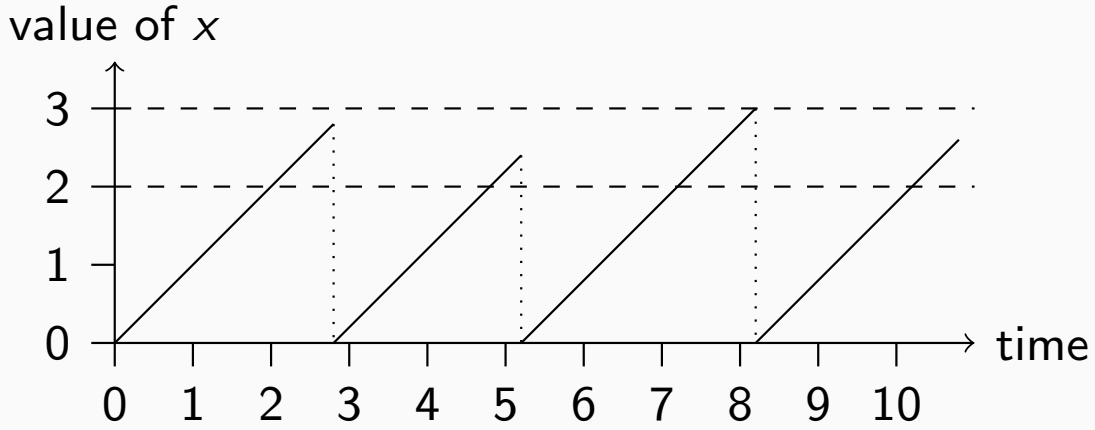
Timed Automata Formally

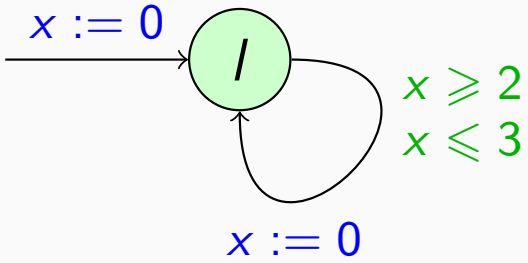
- A *Timed Automaton* is a finite graph with
  - finitely many vertices/locations V, labeled with
    - ★ location invariants,
  - initial location $l_0 \in V$,
  - finitely many edges $E \subseteq V \times V$, labeled with
    - ★ guards,
    - ★ actions,
    - ★ assignments.
- *Guards*: Clock constraint or predicate over data variables
  - Clock constraint: $g ::= x \leqslant n \mid x \geqslant n \mid x < n \mid x > n \mid g \wedge g$
  - Data predicate: "any logical expression" you would write in C
- *Actions*: On channels for synchronization
  - `a?` or `a!`
- *Assignments*: Clock reset or data variable assignment
  - `x:=0`, `i:=i+1`
- *Location Invariants:* Clock constraints, like guards

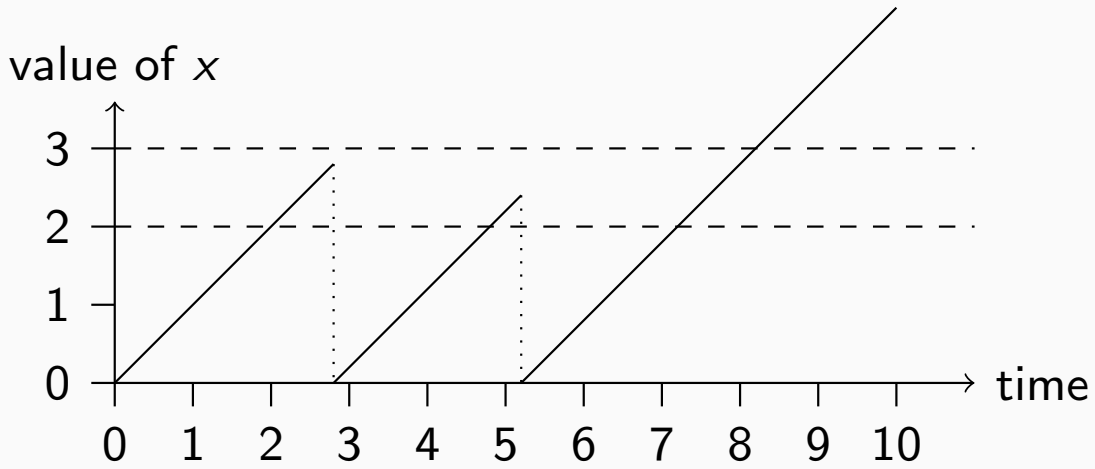
Location Invariants

- Time may only progress as long as invariant satisfied
- Enter a location only if invariant satisfied





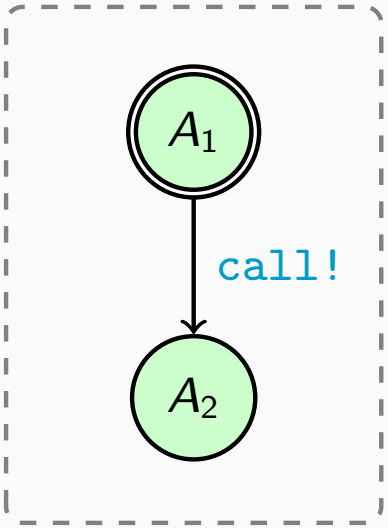


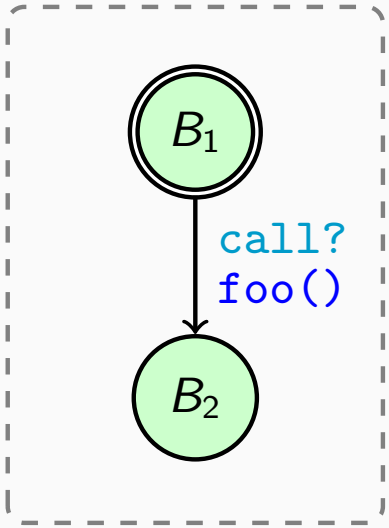


Ada Programs as Timed Automata: Rendezvous

```
task body A is
begin
     ...
     B.Call;
     ...
end A;
```

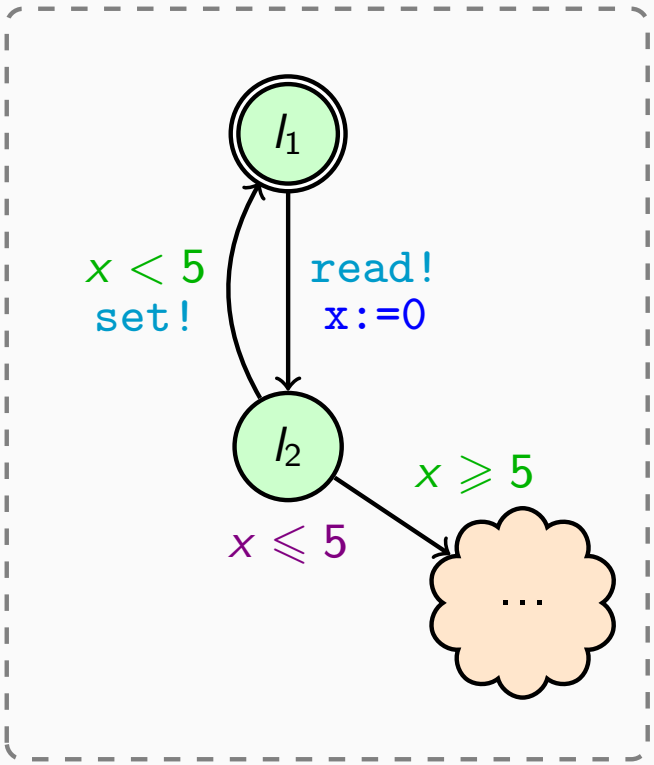
```
task body B is
begin
     ...
     accept Call do
          foo();
     end Call
     ...
end B;
```





Ada Programs as Timed Automata: Timeouts

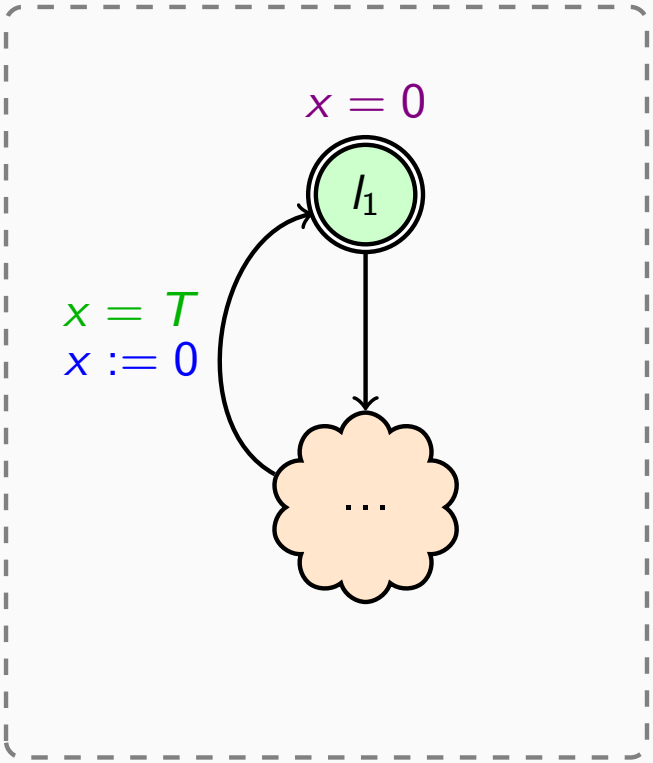
```
loop
    Temperature.read;
    select
        Controller.set;
        or
        delay 5;
        ...
    end select;
end loop;
```



- Message passing via variables

Ada Programs as Timed Automata: Periodic Behaviour

```
task body PeriodicTask is
    T: constant Duration := 5.0;
    Next_Time: Time;
begin
    Next_Time := Clock + T;
    loop
        ...
        delay until Next_Time;
        Next_Time := Next_Time + T;
    end loop;
end PeriodicTask;
```



Fahrplan

1. Motivation
   - Errors in Software Development
   - Modelling and Verification
2. Modelling
   - Finite Automata
   - Timed Automata
   - Ada Programs as Timed Automata
3. Verification
   - Specifications
   - UPPAAL

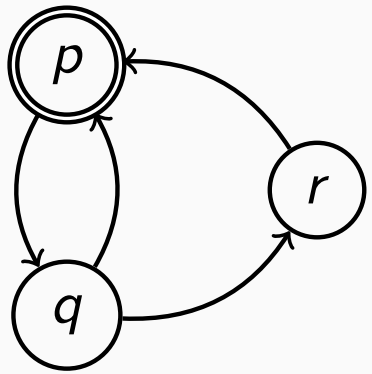
Specifications

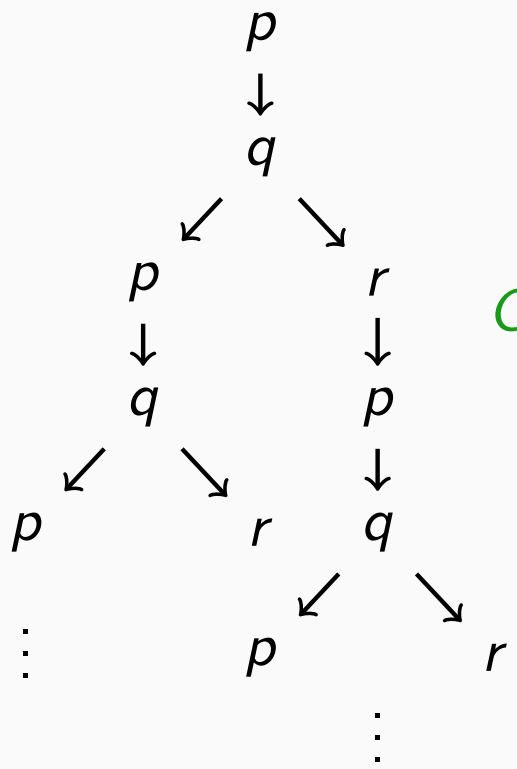
- Specify requirements as *properties*
- Formulated over the state space
- *Safety* properties
  - "Something (bad) should not happen"
  - Method: Reachability analysis
- *Liveness* properties
  - "Something (good) must happen/be repeated"
  - Method: Loop detection

Computation Tree

- Nondeterministic behavior is a *tree*

Automaton





Computation Tree

- Formulate properties over that tree
  - From r, always possible to come back
  - From anywhere, q always reachable
  - Possible to never visit r
  - No deadlock: always a child node
  - ...

Temporal Logic

- Formal way of specifying properties
- *Propositional* state properties:

```
p ::= A.n | gc | gd | not p | p or p | p and p
                   | p imply p
```

`A.n`: Automaton A is in location n
`gc`: Clock constraint, e.g., $x \geqslant 2$
`gd`: Data predicate, e.g., $i == 100$
Rest: Logical connectives (boolean)

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

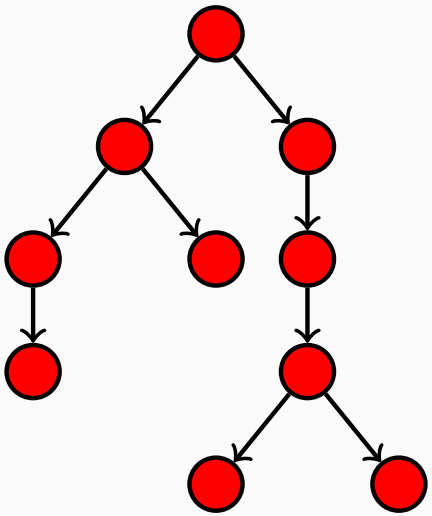
- (UPPAAL cannot nest them)

Operator = Path quantifier + State operator

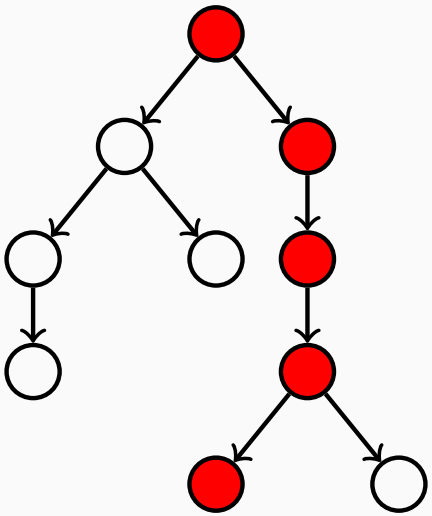
- `A`, `E`: Path quantifiers (**A**lways, **E**ventually)
- `[]`, `<>`: State operators (often written `G`, `F`: **G**lobally, **F**inally)

Temporal Operators

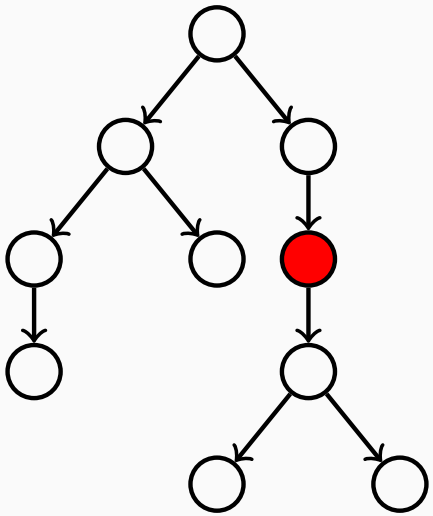
`A[] p`



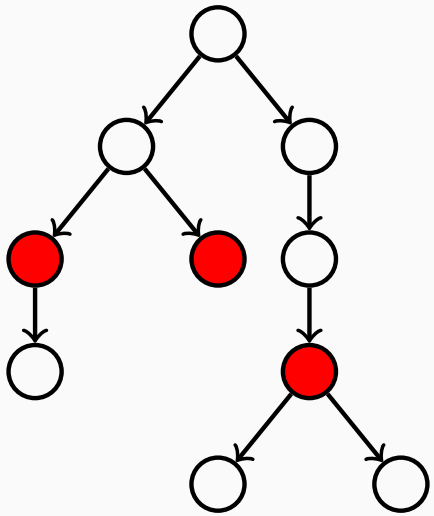
`E[] p`



`E<> p`



`A<> p`



Example Properties

1. It is possible that automaton A reaches location *done*
   - `E<> A.done`
2. Automata A and B are never in locations *critical* at the same time
   - `A[] not (A.critical and B.critical)`
3. It is possible that automaton C never reaches the *reward* location
   - `E[] not C.reward`
4. The system does not deadlock
   - `A[] not deadlock`
5. ...? Whenever automaton B is in location *foo*, clock x is at least 3
   - `A[] B.foo imply x >= 3`
6. ...?No matter what, automaton C eventually reaches location *reward*
   - `A<> C.reward`
7. ...? It is possible that clock x of automaton D is at least 5 but D is not in location *done*
   - `E<> D.x >= 5 and not D.done`

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.