Verification

Lecture 1

Bernd Finkbeiner
Peter Faymonville
Michael Gerke

UNIVERSITÄT
DES
SAARLANDES

Course Meetings

- **Lectures:**
  Tuesdays, 16:15 - 17:55, E1 3 / HS 3
  Thursdays, 14:15 - 15:55, E1 3 / HS 3
- **Tutorials:**
  Mondays, 16:00 - 18:00, E1 3 / SR 015 **or**
  Wednesdays, 12:00 - 14:00, E1 3 / SR 015
- **Office Hours:**
  Bernd Finkbeiner: Wednesdays, 3-4pm, E1 3 / 506
  Peter Faymonville: E1 3 / 533
  Michael Gerke: E1 3 / 507

Problem Sets

- Website:
  `http://react.cs.uni-saarland.de/teaching/`
- Released every Thursday (first on October 20th)
- Due next Thursday, work in groups of up to 3 students
- Submit to our postbox **before** Thursday lecture (or give it to us at the **start** of the lecture)
- **Individual** feedback:
  mandatory discussion slot per group
- Format: 15 minutes, slots on Thursday after the lecture or on Friday 9-11am
- No grading / solutions only presented in tutorials

Exams

- Qualification: Miss at most two discussion slots + hand in solutions to all problem sets
- Three exams: Midterm, Endterm, Final
- Need to pass 2 out of 3 to pass the course
- Grading: average of best 2
- Midterm: 20.12.2011, 4pm
- Endterm: 09.02.2012, 2pm
- Final: end of March (oral/written to be determined)

Administration

Data

- Stammvorlesung, 9 CP
- Bachelor or Master in Computer Science

Registration

- Register on LSF/HISPOS
  `https://lsf.uni-saarland.de`
- Sign up on the paper sheet for tutorial and discussion

Course topic

Algorithms for **automatic verificaton** of hardware and software

- Model checking
- Deductive verification

based on methods from

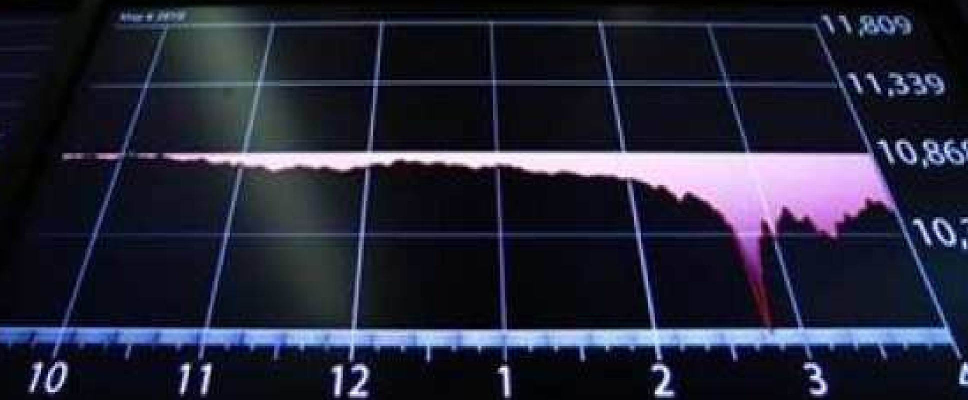
- automata theory
- logic
- symbolic data structures

Connections to other courses

This lecture will provide **foundations** and motivation for the following courses:

- Quantitative Model Checking
- Semantics
- Compiler Construction / Static Analysis
- Automated Reasoning
- Embedded Systems
- Automata, Games, and Verification

MasterCard
VISA
AMERICAN EXPRESS


Buy Imbalance: Shrs
11,809
11,339
10
11
12
1
2
3
DJI
10,519.50
-348.63

The importance of software correctness

- Rapidly increasing **integration of Information and Communication Technology** in different applications:
  - embedded systems
  - communication protocols
  - transportation systems
- Reliability depends on hard- and software **integrity**
- Defects can be **fatal** and extremely **costly**
  - products subject to mass-production
  - safety-critical systems

What is system verification?

System verification amounts to check whether a system fulfills the qualitative requirements that have been identified

Verification ≠ validation:

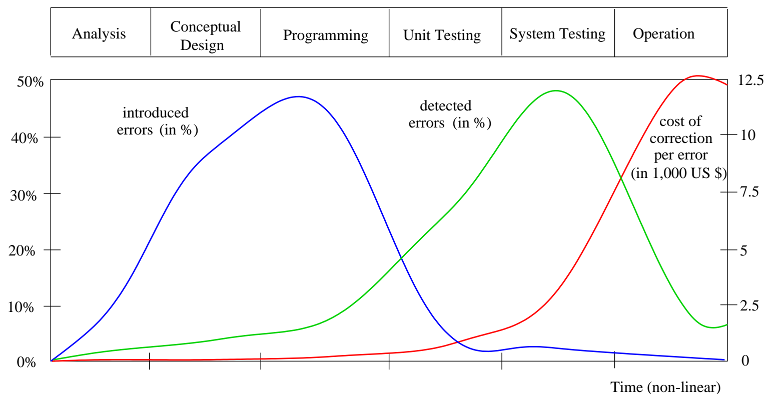
Verification = "check that we are building the thing **right**"

Validation = "check that we are building the **right** thing"

Software verification techniques

- **Peer reviewing**
  - static technique: manual code inspection, no software execution
  - detects between 31 and 93% of defects with median of about 60%
  - subtle errors (concurrency and algorithm defects) hard to catch
- **Testing**
  - dynamic technique in which software is executed
- **Some figures**
  - 30% to 50% of software project costs devoted to testing
  - more time and effort is spent on validation than on construction
  - accepted defect density: about 1 defects per 1,000 code lines

Catching software bugs: the sooner, the better



Formal methods

Formal methods are the

"applied mathematics for modelling and analysing ICT systems"

They offer a large potential for

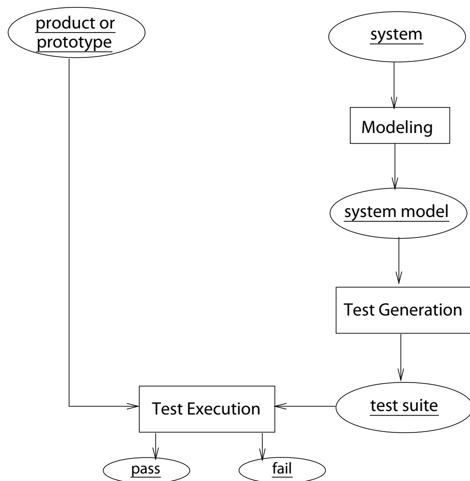
- obtaining an **early integration** of verification in the design process
- providing **more effective** verification techniques (higher coverage)
- **reducing** the verification time

Highly recommended by IEC, ESA, FAA and NASA
for safety-critical software

Formal verification techniques for property ϕ

- **deductive methods**
  - method: provide a formal **proof** that ϕ holds
  - tool: (automated) theorem prover
  - applicable if: system has form of a mathematical theory
- **model checking**
  - method: **systematic check** on ϕ in all states
  - tool: model checker (Spin, NuSMV, UppAal, ...)
  - applicable if: system generates (finite) behavioural model
- **model-based simulation or testing**
  - method: test for ϕ by exploring possible behaviours
  - tool: simulator/tester
  - applicable if: system defines an executable model

Model-based testing



testing/simulation can show the presence of errors,
not their absence

Milestones in formal verification

- Mathematical approach towards program correctness

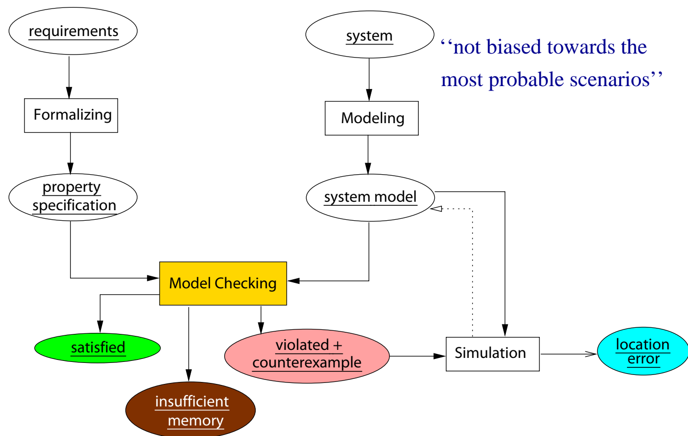
  (Turing, 1949)
- Syntax-based technique for sequential programs (Hoare, 1969)
  - for a given input, does a computer program generate the correct output?
  - based on compositional proof rules expressed in predicate logic
- Syntax-based technique for concurrent programs (Pnueli, 1977)
  - can handle properties referring to situations during the computation
  - based on proof rules expressed in temporal logic
- Automated verification of concurrent programs

  (Emerson, Clarke, Sifakis 1981)
  - model-based instead of proof-rule based approach
  - does the concurrent program satisfy a given (logical) property?

Model checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

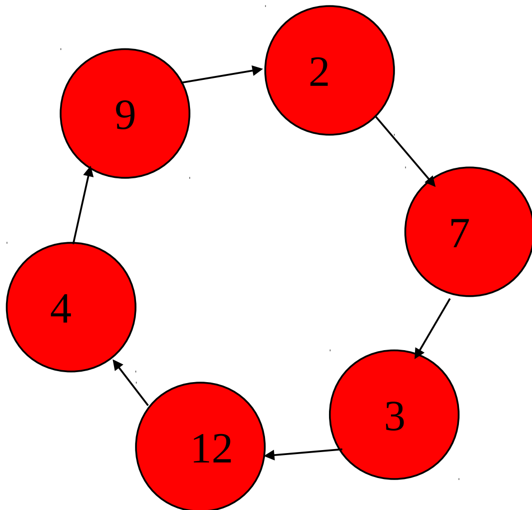
Model checking overview



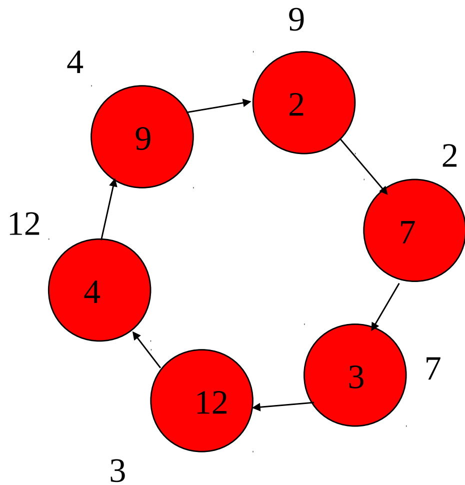
‘‘not biased towards the most probable scenarios’’

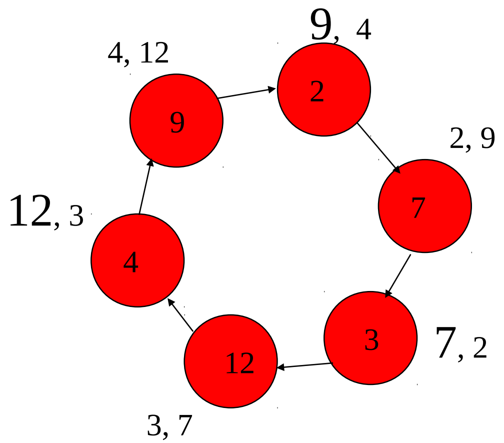
Example: Leader Election

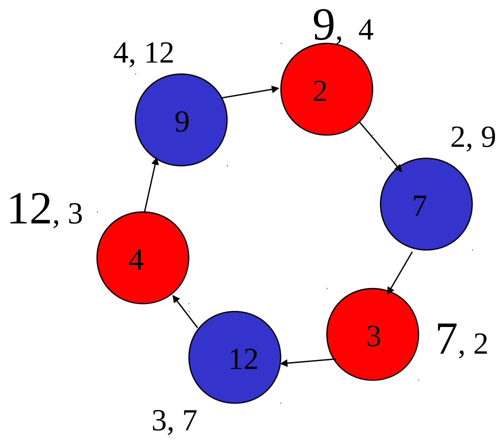
A directed ring of computers. Each has a unique value. Communication is from left to right. Find out which value is the greatest.

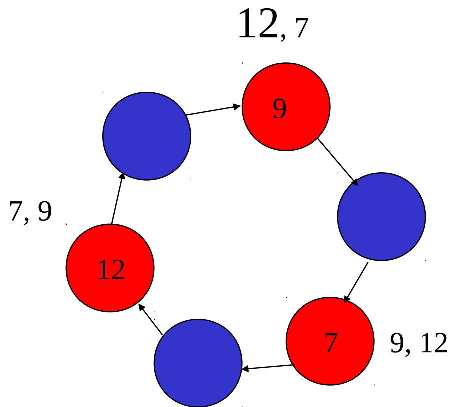


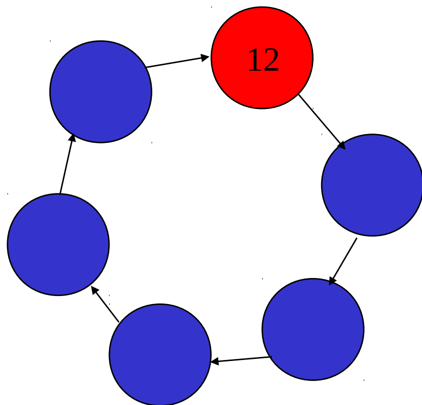
- Initially, all the processes are active.
- A process that finds out it does not represent a value that can be maximal turns to be passive.
- A passive process just transfers values from left to right.
- The algorithm executes in phases.
- In each phase, each process first sends its current value to the right.
- Each process, when receiving the first value from its left compares it to its current value.
  - If same: this is the maximum. Tell others.
  - Not same: send current value again to right.
- When receiving the second value: compare the three values received. These are values
  - of the process itself.
  - of the left active process.
  - of the second active process on the left.
- If the left active process has greatest value among three, then keep this value. Otherwise, become passive.


9
4
9
2
2
7
12
4
3
7
12
3


9, 4
4, 12
2
9
2, 9
12, 3
7
4
3
7, 2
12
3, 7


9, 4
4, 12
2
9
2, 9
7
12, 3
4
3
7, 2
12
3, 7


12, 7
9
7, 9
12
7
9, 12


12

```
#define N 5       /* number of processes */
#define I 3       /* node given the smallest number */
#define L 10      /* size of buffer (>= 2*N) */
mtype = { one, two, winner };
chan q[N] = [L] of { mtype , byte };
byte nr_leaders = 0;
proctype node (chan in, out; byte mynumber) {
   bit Active = 1, know_winner = 0;
   byte rec, maximum = mynumber, neighbor;
   printf("MSC: %d\n", mynumber);
   out!one(mynumber);
end:    do
   :: in?one(rec) ->
      if
      :: Active ->
         if
         :: rec != maximum ->
            out!two(rec);
            neighbor = rec
         :: else ->
            assert(rec == N); /* max is greatest number */
            know_winner = 1;
            out!winner(rec);
         fi
      :: else ->
         out!one(rec)
      fi
   :: in?two(rec) ->
      if
      :: Active ->
         if
         :: neighbor > rec && neighbor > maximum ->
            maximum = neighbor;
            out!one(neighbor)
         :: else ->
            Active = 0
         fi
      :: else ->
         out!two(rec)
      fi
   :: in?winner(rec) ->
      if
      :: rec != mynumber ->
         printf("MSC: LOST\n");
      :: else ->
         printf("MSC: LEADER\n");
         nr_leaders++;
         assert(nr_leaders == 1)
      fi ;
      if
      :: know_winner
      :: else -> out!winner(rec)
      fi ;
      break
   od
}

init {
   byte proc;
   atomic {
      proc = 1;
      do
      :: proc <= N ->
         run node (q[proc-1], q[proc%N], (N+I-proc)%N+1);
         proc++
      :: proc > N ->
         break
      od
   }
}
```

Proving Assertions

- inline assertions

```
        :: in?winner(rec) ->
           if
           :: rec != mynumber ->
              printf("MSC: LOST\n");
           :: else ->
              printf("MSC: LEADER\n");
              nr_leaders++;
              assert(nr_leaders == 1)
           fi ;
           if
           :: know_winner
           :: else -> out!winner(rec)
           fi ;
           break
        od
     }
```

- run a monitor process

```
proctype monitor(){
   assert( nr_leaders <= 1 )
}
```

- prove a property given in temporal logic

The pros of model checking

- widely applicable (hardware, software, protocol systems, ...)
- allows for partial verification (only most relevant properties)
- potential “push-button” technology (software-tools)
- rapidly increasing industrial interest
- in case of property violation, a counter-example is provided
- sound and interesting mathematical foundations
- not biased to the most possible scenarios (such as testing)

The cons of model checking

- mainly focused on control-intensive applications (less data-oriented)
- any validation model checking is only as "good" as the system model
- no guarantee about completeness of results
- impossible to check generalisations (in general)

Deductive Verification

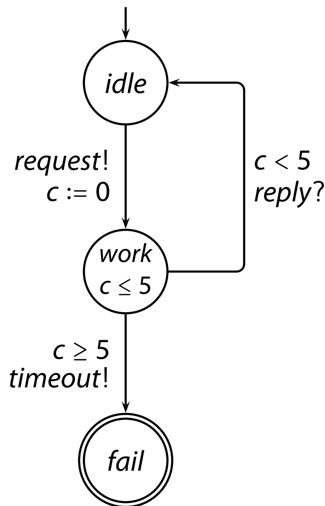
```
method isqrt(N : int) returns (R : int)
       requires N >= 0 ;
       ensures  (R + 1) * (R + 1) > N ;
       ensures  R * R <= N ;
{
   R := 0 ;
       while ((R + 1) * (R + 1) <= N)
       {
               R := R + 1 ;
       }
}
```

Deductive Verification

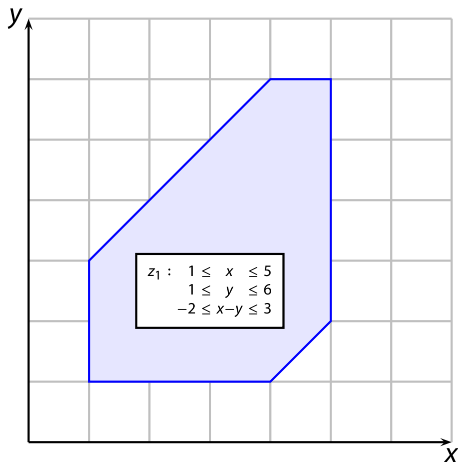
```
method isqrt(N : int) returns (R : int)
       requires N >= 0 ;
       ensures  (R + 1) * (R + 1) > N ;
       ensures  R * R <= N ;
{
   R := 0 ;
       while ((R + 1) * (R + 1) <= N)
               invariant R * R <= N ;
       {
               R := R + 1 ;
       }
}
```

Timed Automata

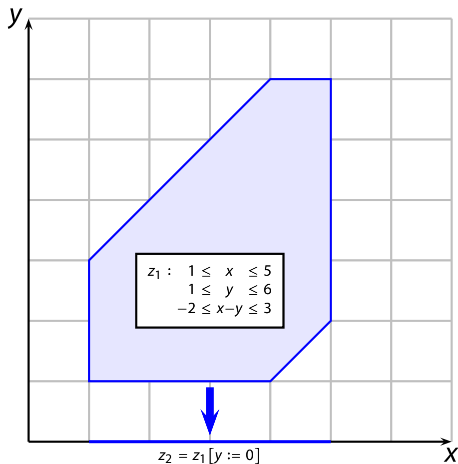
- Finite-state automata + clocks
- Locations with invariants
- Transitions:
  - guard
  - synchronization label
  - clock resets
- state = location + clock valuation
  → infinitely many states!
  → idea: finite number of equivalence classes



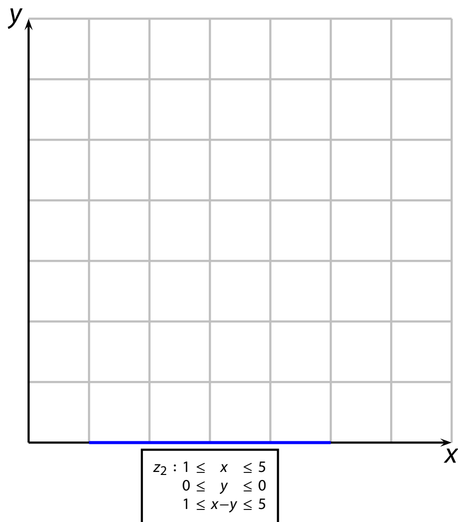
Clock zones: abstraction for timed automata



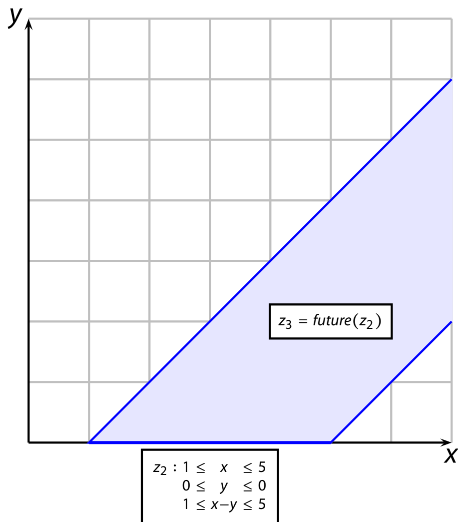
Clock zones: abstraction for timed automata



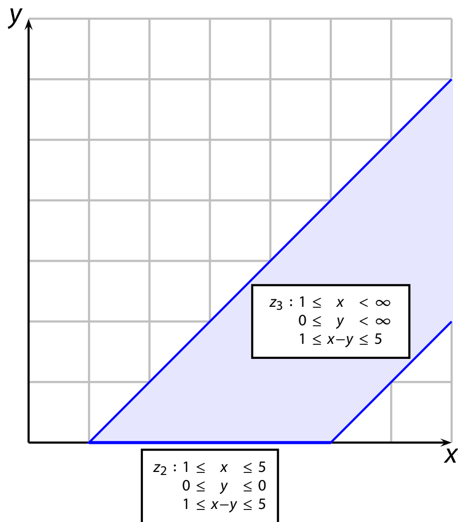
Clock zones: abstraction for timed automata



Clock zones: abstraction for timed automata

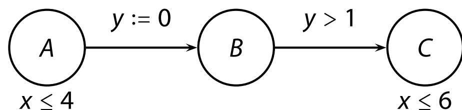


Clock zones: abstraction for timed automata

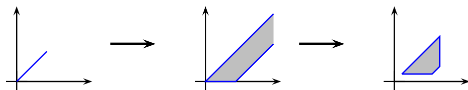


Clock zones: abstraction for timed automata

- timed automaton



- zone graph



- abstract states: (q, z) -- finite number of states!
- reachability in timed automata is **decidable**.

Course content

- **Modeling** hard- and software systems
- **Linear-time** model checking
- **Branching-time** model checking
- Equivalences and **abstraction**
- **Deductive** verification
- **Real-time** systems

Transition Systems

Transition systems

- model to describe the behaviour of systems
- digraphs where nodes represent states, and edges model transitions
- state:
  - the current colour of a traffic light
  - the current values of all program variables + the program counter
  - the current value of the registers together with the values of the input bits
- transition: ("state change")
  - a switch from one colour to another
  - the execution of a program statement
  - the change of the registers and output bits for a new input

Transition systems

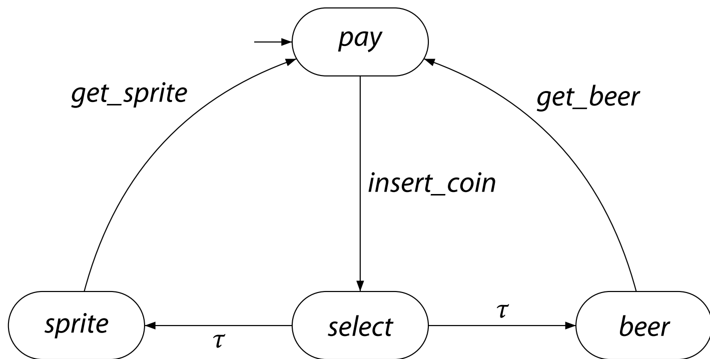
A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states
- Act is a set of actions
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ is a set of initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labeling function

S and Act are either finite or countably infinite

Notation: $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \longrightarrow$

A beverage vending machine



Direct successors and predecessors

$$Post(s,\alpha) = \left\{ s' \in S \mid s \xrightarrow{\alpha} s' \right\}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s,\alpha)$$

$$Pre(s,\alpha) = \left\{ s' \in S \mid s' \xrightarrow{\alpha} s \right\}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s,\alpha).$$

$$Post(C,\alpha) = \bigcup_{s \in C} Post(s,\alpha), \quad Post(C) = \bigcup_{s \in C} Post(s) \text{ for } C \subseteq S.$$

$$Pre(C,\alpha) = \bigcup_{s \in C} Pre(s,\alpha), \quad Pre(C) = \bigcup_{s \in C} Pre(s) \text{ for } C \subseteq S.$$

State s is called terminal if and only if $Post(s) = \varnothing$

Action- and *AP*-determinism

Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is action-deterministic iff:

$$|I| \leq 1 \quad \text{and} \quad |Post(s, \alpha)| \leq 1 \quad \text{for all } s, \alpha$$

Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is *AP*-deterministic iff:

$$|I| \leq 1 \ \text{ and } \ |\underbrace{Post(s) \cap \{ s' \in S \mid L(s') = A \}}_{\text{equally labeled successors of } s}| \leq 1 \quad \text{for all } s, A \in 2^{AP}$$

The role of nondeterminism

Here: nondeterminism is a feature!

- to model concurrency by interleaving
  - no assumption about the relative speed of processes
- to model implementation freedom
  - only describes what a system should do, not how
- to model under-specified systems, or abstractions of real systems
  - use incomplete information

in automata theory, nondeterminism may be exponentially more succinct

but that's not the issue here!

Executions

- A finite execution fragment ρ of TS is an alternating sequence of states and actions ending with a state:

$$\rho \;=\; s_0\,\alpha_1\,s_1\,\alpha_2\,\ldots\alpha_n\,s_n \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i < n.$$

- An infinite execution fragment ρ of TS is an infinite, alternating sequence of states and actions:

$$\rho \;=\; s_0\,\alpha_1\,s_1\,\alpha_2\,s_2\,\alpha_3\ldots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i.$$

- An execution of TS is an initial, maximal execution fragment
  - a maximal execution fragment is either finite ending in a terminal state, or infinite
  - an execution fragment is initial if $s_0 \in I$

Example executions

$$\rho_1 = pay \xrightarrow{coin} select \xrightarrow{\tau} sprite \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} sprite \xrightarrow{sget} \ldots$$

$$\rho_2 = select \xrightarrow{\tau} sprite \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} beer \xrightarrow{bget} \ldots$$

$$\rho = pay \xrightarrow{coin} select \xrightarrow{\tau} sprite \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} sprite$$

Execution fragments ρ_1 and ρ are initial, but ρ_2 is not
ρ is not maximal as it does not end in a terminal state
Assuming that ρ_1 and ρ_2 are infinite, they are maximal

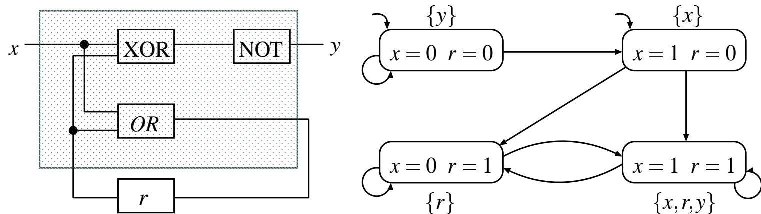
Reachable states

State $s \in S$ is called reachable in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s .$$

$Reach(TS)$ denotes the set of all reachable states in TS.

Modeling sequential circuits



Transition system representation of a simple hardware circuit
Input variable x, output variable y, and register r
Output function $\neg(x \oplus r)$ and register evaluation function $x \vee r$

Atomic propositions

Consider two possible state-labelings:

- Let $AP = \{ x, y, r \}$
  - $L(\langle x = 0, r = 1 \rangle) = \{ r \}$ and $L(\langle x = 1, r = 1 \rangle) = \{ x, r, y \}$
  - $L(\langle x = 0, r = 0 \rangle) = \{ y \}$ and $L(\langle x = 1, r = 0 \rangle) = \{ x \}$
  - property e.g., "once the register is one, it remains one"
- Let $AP' = \{ x, y \}$ -- the register evaluations are now "invisible"
  - $L(\langle x = 0, r = 1 \rangle) = \varnothing$ and $L(\langle x = 1, r = 1 \rangle) = \{ x, y \}$
  - $L(\langle x = 0, r = 0 \rangle) = \{ y \}$ and $L(\langle x = 1, r = 0 \rangle) = \{ x \}$
  - property e.g., "the output bit y is set infinitely often"