Verification

Lecture 1

Bernd Finkbeiner

UNIVERSITÄT DES SAARLANDES

Team

- Lectures: Bernd Finkbeiner, Martin Zimmermann
- Exercises: Leander Tentrup
- Discussion slots: Peter Faymonville, Michael Gerke, Felix Klein, Andrey Kupriyanov, Heinrich Ody, Markus Rabe, Hazem Torfah

Structure

	Mon	Tue	Wed	Thu	Fri
9:00--10:00	lecture	lecture	lecture	lecture	lecture
10:00--11:00	group work	group work	group work	group work	group work
11:00--11:30	discussion	discussion	discussion	discussion	discussion
					BBQ
14:00--15:00	lecture	lecture		lecture	lecture
15:00--16:00	group work	group work		group work	group work
16:00--16:30	discussion	discussion		discussion	discussion

Groups

- Please sign up in groups of 3
- Each group has an assigned meeting room and an assigned tutor
- You meet with your tutor during your “discussion slot”

Exams

- **Please register for the exam in LSF/HISPOS**
  `https://lsf.uni-saarland.de`
- Exam: 09.10.2013, 9am
- Backup Exam: TBA
- The grade solely depends on the performance in the exam.
- You are allowed to take part in the exam if you reach at least 50% of the total points in the assignments presented in the discussion slots.

Course topic

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

based on methods from

- automata theory
- logic

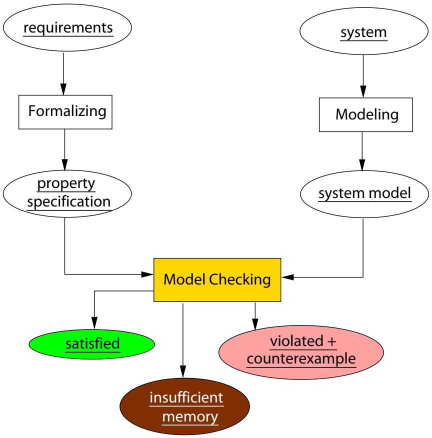
Early history

- Mathematical approach towards program correctness
  (Turing, 1949)
- Proof rules for sequential programs (Hoare, 1969)
  - for a given input, does a computer program generate the correct output?
  - based on proof rules expressed in predicate logic
- Proof rules for concurrent programs (Pnueli, 1977)
  - does the program perform correctly over an infinite run?
  - based on proof rules expressed in temporal logic
- Automated verification of concurrent programs
  (Emerson, Clarke, Sifakis 1981)
  - systematic state space traversal
  - “model checking”

Model checking

Model checking is an automated technique that, given a model of a system and a formal property, systematically checks whether this property holds for that model.

Model checking overview



Course structure

- Week 1: Hardware model checking
  VIS, CTL, CTL model checking, BDDs, LTL
- Week 2: Protocol verification
  SPIN, LTL model checking, bounded model checking
- Week 3: Real-time systems
  Uppaal, timed automata, DBMs, bisimulation
- Week 4: Software verification
  PiVC, deductive verification, decision procedures

Plan for today

- The VIS model checker
- Verilog examples
- Transition systems
- CTL

VIS

- VIS: "Verification interacting with synthesis"
- verification and synthesis system for finite-state hardware systems
- developed at University of California, Berkeley, and University of Colorado, Boulder
- system given in (subset of) Verilog
- available from `http://vlsi.colorado.edu/~vis/` or as a convenient VirtualBox Appliance from the course webpage

Counter

```
module counter(clk, count);

    input clk;
    output count;
    wire clk;                  ← wire = connector
    reg [1:0] count;           ← reg = register (memory)

    initial begin              ← initialization
      count = 0;
    end

    always @(posedge clk)      ← executed in every step
      count = count + 1;

endmodule
```

Simulation

```
vis> read_verilog counter.v
counter.v
vis> init_verify
vis> sim -n 5
```

```
# vis release 2.4 (compiled Mo 2.  Sep 09:09:38 CEST 2013)
# Network:  counter
# Simulation vectors have been randomly generated

.inputs
.latches count<0> count<1>
.outputs count<0> count<1>
.initial 0 0

.start_vectors

# ; count<0> count<1> ; count<0> count<1>

; 0 0 ; 0 0
; 1 0 ; 1 0
; 0 1 ; 0 1
; 1 1 ; 1 1
; 0 0 ; 0 0
# Final State :  0 0
```

3-bit counter

```
module counter(clk);
    input clk;
    wire clk;
    wire [2:0] count;

    counter_cell bit0 (clk, 1, count[0]);
    counter_cell bit1 (clk, count[0], count[1]);
    counter_cell bit2 (clk, count[1], count[2]);

endmodule
```

3-bit counter

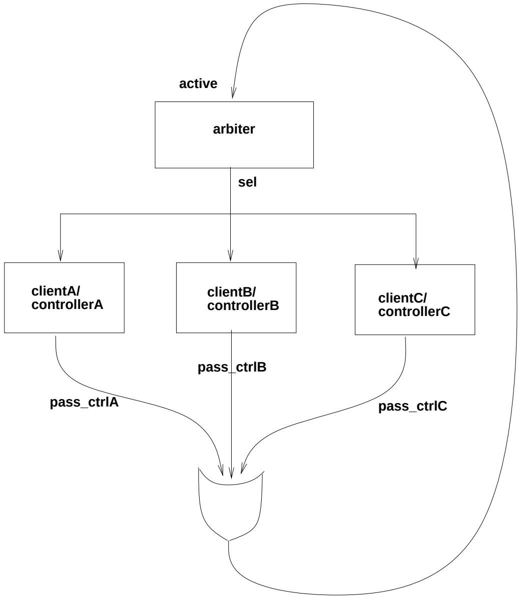
```
module counter_cell(clk, carry_in, carry_out);
    input clk;
    input carry_in;
    output carry_out;
    reg value;
                    ↓ continuous assignment
    assign carry_out = value & carry_in;
    initial value = 0;

    always @(posedge clk) begin
       case(value)
           0:  value = carry_in;
           1:  if (carry_in ==0) value = 1;
           else value = 0;
       endcase
    end
endmodule
```

Arbiter

- Three requesting modules (called clients) are competing to get a bus access.
- At any point, only one module is allowed to get a bus access.
- Each client has a controller attached to it, from which an acknowledgment is given.
- All the controllers communicate with an arbiter so that at any time at most one controller gives an acknowledgment.

(See vis-2.4/examples/arbiter)

Arbiter


active
arbiter
sel
clientA/ controllerA
clientB/ controllerB
clientC/ controllerC
pass_ctrlA
pass_ctrlB
pass_ctrlC

```
typedef enum A, B, C, X selection;
typedef enum IDLE, READY, BUSY controller_state;
typedef enum NO_REQ, REQ, HAVE_TOKEN client_state;

module main(clk);
   input clk;
   output ackA, ackB, ackC;
   selection wire sel;
   wire active;

   assign active = pass_tokenA || pass_tokenB || pass_tokenC;

   controller controllerA(clk, reqA, ackA, sel, pass_tokenA, A);
   controller controllerB(clk, reqB, ackB, sel, pass_tokenB, B);
   controller controllerC(clk, reqC, ackC, sel, pass_tokenC, C);
   arbiter arbiter(clk, sel, active);
   client clientA(clk, reqA, ackA);
   client clientB(clk, reqB, ackB);
   client clientC(clk, reqC, ackC);

endmodule
```

```
module controller(clk, req, ack, sel, pass_token, id);
   input clk, req, sel, id;
   output ack, pass_token;

   selection wire sel, id;
   reg ack, pass_token;
   controller_state reg state;

   initial state = IDLE;
   initial ack = 0;
   initial pass_token = 1;

   wire is_selected;
   assign is_selected = (sel == id);
```

```
always @(posedge clk) begin
      case(state)
      IDLE:
          if (is_selected)
             if (req)
             begin
                 state = READY;
                 pass_token = 0;
             end
             else
                 pass_token = 1;
          else
             pass_token = 0;
```

```
            READY:
                begin
                   state = BUSY;
                   ack = 1;
                end
            BUSY:
                if (!req)
                begin
                   state = IDLE;
                   ack = 0;
                   pass_token = 1;
                end
            endcase
      end
   endmodule
```

```
module arbiter(clk, sel, active);
    input clk, active;
    output sel;

    selection wire sel;
    selection reg state;

    initial state = A;

    assign sel = active ?  state :  X;

    always @(posedge clk) begin
           if (active)
                case(state)
                   A: state = B;
                   B: state = C;
                   C: state = A;
                endcase
    end
endmodule
```

```
module client(clk, req, ack);
   input clk, ack;
   output req;

   reg req;
   client_state reg state;

   wire rand_choice;

   initial req = 0;
   initial state = NO_REQ;

   assign rand_choice = $ND(0,1);

   always @(posedge clk) begin
         case(state)
             NO_REQ:
                if (rand_choice)
                begin
                   req = 1;
                   state = REQ;
                end
```

```
          REQ:
             if (ack) state = HAVE_TOKEN;
          HAVE_TOKEN:
             if (rand_choice)
             begin
                req = 0;
                state = NO_REQ;
             end
       endcase
   end
endmodule
```

Model checking

- Mutual exclusion: No two different acks are given at the same time.
  ```
  AG ( !(ackA=1 * ackB=1 + ackB=1 * ackC=1 +
  ackC=1 * ackA=1) );
  ```

```
vis> read_verilog arbiter.v
vis> init_verify
vis> model_check arbiter.ctl

# MC: formula passed -- AG(!((((ackA=1 *
ackB=1) + (ackB=1 * ackC=1)) + (ackC=1 *
ackA=1))))
```

Transition systems

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

Transition systems

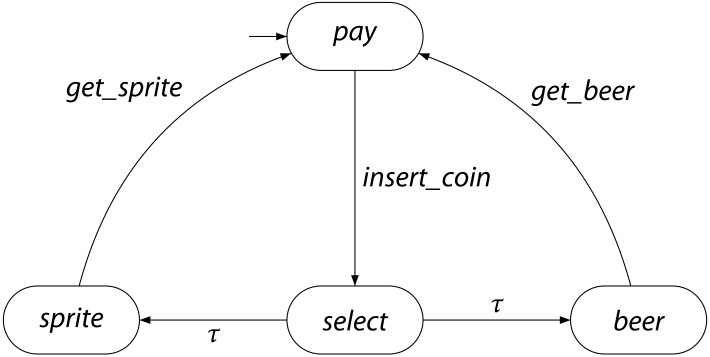
A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- 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

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

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

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

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

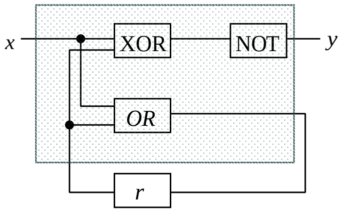
Reachable states

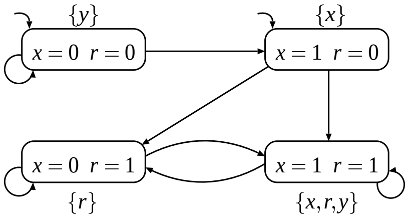
State $s \in S$ is called reachable in TS if there exists an initial, finite execution fragment

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

$Reach(TS)$ 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 $x \vee r$