Embedded Systems
Part 2

Barbara Jobstmann
CNRS/Verimag
General

- Lecture + exercise session
- Presence sheet
- Homework
- Questions: barbara.jobstmann@imag.fr

TODO: send me an email saying that you attended my course
Goal: Analyze Reactive Systems

- Reactive systems?
Goal: Analyze Reactive Systems

• Reactive systems
  – Synchronously interact with environment using input and output signals
  – In every step: input, output, and state variables are updated \textit{synchronously}
  – E.g., hardware, controllers,…

• Other systems/programs, e.g., transformational programs, event-based interactive systems (async.)
Overview: Analyzer

Build Model:
give precise semantics (meaning) to what a system does

Annotate model with specification

Analyze model (model checking)

Correct (Proof)

Incorrect (Counter-example)
Overview: Analyzer

- **Build Model**: give precise semantics to what a system does
- **Annotate model with specification**
- **Specification**: Source code
- **Analyze model** (model checking)
  - Correct (Proof)
  - Incorrect (Counter-example)
- **Subset of Verilog (HDL)**
- **Temporal Logics**
- **Transition system (FSA and Boolean formulas)**
- **Mark states**
- **Search algorithms (reach bad states or circles)**
- **Invariant**
- **CEX**
Outline of Today

• Examples:
  – Verilog (subset)
  – Assertions
  – Temporal properties

• Computational Tree Logic (CTL)
  – Syntax
  – Semantics
module counter (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire clock;
    wire enable;
    reg [1:0] count;

    initial begin
        count = 0;
    end

    always @(posedge clock) begin
        if (enable)
            count = count + 1;
        else
            count = count;
    end
endmodule // counter
module counter (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire clock;
    wire enable;
    reg [1:0] count;

    initial begin
        count = 0;
    end

    always @(posedge clock) begin
        if (enable)
            count = count + 1;
        else
            count = count;
    end
endmodule

// counter

2-Bit Counter
module counter (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire clock;
    wire enable;
    reg [1:0] count;

    initial begin
        count = 0;
    end

    always @(posedge clock) begin
        if (enable)
            count = count + 1;
        else
            count = count;
    end
endmodule // counter
2-Bit Counter

module counter (clock, enable, count);
  input clock;
  input enable;
  output count;
  wire clock;
  wire enable;
  reg [1:0] count;

  initial begin
    count = 0;
  end

  always @(posedge clock) begin
    if (enable)
      count = count + 1;
    else
      count = count;
  end
endmodule // counter

Variables: two types
(1) wires (connectors)
(2) registers (memory)

Def: type bit-width name

wire [3:0] mywire;
  ...mywire[0], mywire[1],..
typedef enum{a,b,c} stateT;
  stateT reg myState;
module counter (clock, enable, count);
  input clock;
  input enable;
  output count;
  wire clock;
  wire enable;
  reg [1:0] count;

  initial begin
    count = 0;
  end

  always @(posedge clock) begin
    if (enable)
      count = count + 1;
    else
      count = count;
  end
endmodule // counter
module counter (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire    clock;
    wire    enable;
    reg [1:0] count;

initial begin
    count = 0;
end

always @(posedge clock) begin
    if (enable)
        count = count + 1;
    else
        count = count;
end
endmodule // counter

Behavior at each time step (synchronous system!)
module counter (clock, enable, count);
  input clock;
  input enable;
  output count;
  wire clock;
  wire enable;
  reg [1:0] count;

initial begin
  count = 0;
end

always @(posedge clock) begin
  if (enable)
    count = count + 1;
  else
    count = count;
end
endmodule // counter

<table>
<thead>
<tr>
<th>Time</th>
<th>Input enable</th>
<th>count[1]</th>
<th>count[0]</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Bit Counter

module counter (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire clock;
    wire enable;
    reg [1:0] count;

    initial begin
        count = 0;
    end

    always @(posedge clock) begin
        if (enable)
            count = count + 1;
        else
            count = count;
    end
endmodule // counter

<table>
<thead>
<tr>
<th>Time</th>
<th>Input enable</th>
<th>count[1]</th>
<th>count[0]</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>
2-Bit Counter

module counter (clock, enable, count);
  input clock;
  input enable;
  output count;
  wire clock;
  wire enable;
  reg [1:0] count;

  initial begin
    count = 0;
  end

  always @(posedge clock) begin
    if (enable)
      count = count + 1;
    else
      count = count;
  end
endmodule // counter

<table>
<thead>
<tr>
<th>Time</th>
<th>Input enable</th>
<th>count[1]</th>
<th>count[0]</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>

Synchronous!

Barbara Jobstmann
module counter (clock, enable, max);
  input clock;
  input enable;
  output max;
  wire clock;
  wire enable;
  wire max;
  reg [1:0] count;

assign max = count[1] && count[0];

initial begin
  count = 0;
end
always @(posedge clock) begin
  if (enable)
    count = count + 1;
  else
    count = count;
end
endmodule // counter

Barbara Jobstmann

Boolean Algebra (or Logic)
Transition System

All Behaviors

Barbara Jobstmann
Transition System
module counter (clock, enable, count);
  input clock;
  input enable;
  output count;
  wire clock;
  wire enable;
  reg [2:0] count;

initial begin
  count = 0;
end

always @(posedge clock) begin
  if (enable) begin
    count = count + 1;
    count = count + 2;
  end else
    count = count;
end
endmodule // counter

Transition System?
4-Bit Counter

```verilog
module counter4bit (clock, enable, count);
  input clock;
  input enable;
  output count;
  wire clock;
  wire enable;
  wire [3:0] count;
  wire enable1;

  assign enable1 = count[1] && count[0];

  counter C0(clock, enable, count[1:0]);
  counter C1(clock, enable1, count[3:2]);
endmodule // counter

`include "counter.v"
```
4-Bit Counter

module counter4bit (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire clock;
    wire enable;
    wire [3:0] count;
    wire enable1;

    assign enable1 = count[1] && count[0];

    counter C0(clock, enable, count[1:0]);
    counter C1(clock, enable1, count[3:2]);
endmodule // counter

`include "counter.v"
module counter4bit (clock, enable, count);
    input clock;
    input enable;
    output count;
    wire clock;
    wire enable;
    wire [3:0] count;
    wire enable1;

    assign enable1 = count[1] && count[0];

    counter C0(clock, enable, count[1:0]);
    counter C1(clock, enable1, count[3:2]);
endmodule // counter

`include "counter.v"

way to include modules of other files
Model Checker VIS

• Model Checker developed by University of Colorado
  • http://vlsi.colorado.edu/~vis/

• Input:
  – Subset of Verilog (we have just seen)
  – First module in file is considered top module
Simulation Demo

• Start vis: vis

```bash
vis> read_verilog counter.v
counter.v
vis>
```

• Load file: read_verilog

```bash
vis> read_verilog counter.v
counter.v
vis>
```
Simulation Demo

• Initialize vis: init_verify
  Shortcut for the 3 commands: flatten_hierarchy; static_order; build_partition_mdds (see help init_verify)
Simulation Demo

• Simulate with random inputs:
  `sim -n 5`

```
vis> sim -n 5
# vis release 2.3 (compiled 13-Jun-10 at 4:40 PM)
# Network: counter
# Simulation vectors have been randomly generated

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

.start_vectors

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

1 ; 0 0 ; 0 0
1 ; 1 0 ; 1 0
0 ; 0 1 ; 0 1
1 ; 0 1 ; 0 1
1 ; 1 1 ; 1 1
# Final State : 1 1
vis>  
```
Example: Arbiter and Clients

Client 0

Client 1

Client n

Arbiter

req[0] → ack[0]


req[n] → ack[n]
Synchronous Arbiter

• Combines daisy-chain and token-ring approach

• Arbiter consists of set of cells
  – One cell for each client
Token Ring

- req
- tokenIn
- clock
- tokenOut
- and
- ack

Barbara Jobstmann
Token Ring
Daisy Chain

Barbara Jobstmann
Note Cell 0 has higher priority than Cell 1.
Synchronous Arbiter

- Main idea: combine daisy-chain approach with token ring
  - If there are few requests, then fixed priority scheme
  - If requests are pending, then wait until token comes around
  - If there are many requests, then follow token-ring approach
Cell

McMillan [McM94]

Barbara Jobstmann
module cell(clock, req, ack, tokenIn, tokenOut, tokenInit, grantIn, grantOut, overIn, overOut);

input clock, req, tokenIn, tokenInit, grantIn, overIn;
output ack, tokenOut, grantOut, overOut;
wire clock, req, tokenIn, tokenInit, grantIn, overIn;
wire ack, tokenOut, grantOut, overOut;
reg token;
reg waiting;
wire tw;

initial begin
  token = tokenInit;
  waiting = 0;
end

assign tw = token & waiting;
assign tokenOut = token;
assign grantOut = ~req & grantIn;
assign overOut = overIn | tw;
assign ack = req & (grantIn | tw);

always @(posedge clock) begin
  waiting = req & (tokenOut | waiting);
  token = tokenIn;
end
endmodule // cell
module cell(clock, req, ack, tokenIn, tokenOut, tokenInit, grantIn, grantOut, overIn, overOut);

input clock, req, tokenIn, tokenInit, grantIn, overIn;
output ack, tokenOut, grantOut, overOut;
wire clock, req, tokenIn, tokenInit, grantIn, overIn;
wire ack, tokenOut, grantOut, overOut;
reg token;
reg waiting;
wire tw;

initial begin
  token = tokenInit;
  waiting = 0;
end

assign tw = token & waiting;
assign tokenOut = token;
assign grantOut = ~req & grantIn;
assign overOut = overIn | tw;
assign ack = req & (grantIn | tw);

always @(posedge clock) begin
  waiting = req & (tokenOut | waiting);
  token = tokenIn;
end
endmodule // cell

~ negation
& and | are bit-wise operations
Arbiter for Two Clients

Note Cell 0 has higher priority than Cell 1.
module arbiter(clk, req, ack);
    input clk, req;
    output ack;
    wire [1:0] req;
    wire [1:0] ack;
    wire t01, t10, o1, o0, g0, g1, dc;
    assign g0 = ~o0;

    wire zero, one;
    assign zero = 1'b0;
    assign one = 1'b1;

    reg [1:0] lreq;
    initial lreq=2'b0;
    always @(posedge clock) lreq=req;

    cell C0(clk, lreq[0], ack[0], t01, t10, one, g0, g1, o1, o0);
    cell C1(clk, lreq[1], ack[1], t10, t01, zero, g1, dc, zero, o1);
endmodule // arbiter

`include "cell.v"
Transition System

Barbara Jobstmann

42
Properties

• Mutual exclusion
  - \( \text{AG}( \neg (\text{ack}[0]=1 \land \text{ack}[1]=1) ) \);
  - In all states reachable from initial state one of the two acknowledge must be low.
  - AG... globally on all possible paths

• Invariant property: a Boolean combination of atomic proposition that must be true in all states

• Invariant are safety properties: something bad never happens
Properties

• Two other properties:
  – $\text{AG}(\text{AF}(\text{lreq}[0]=1 \rightarrow \text{ack}[0]=1));$
  – $\text{AG}(\text{AF}(\text{lreq}[1]=1 \rightarrow \text{ack}[1]=1));$
  – From every state reachable from the initial state, we must eventually see a state in which either there is no request, or there is an acknowledge.
  – $\text{AF}...\text{eventually on all possible paths}$

• Liveness properties: something good happens eventually
Properties

• No acknowledge without request
  - $\text{AG}(\text{ack}[0]=1 \rightarrow \text{lreq}[0]=1)$;
  - $\text{AG}(\text{ack}[1]=1 \rightarrow \text{lreq}[1]=1)$;
  - …an all paths, whenever we encounter an acknowledge then there is also a request.

• Safety or Liveness?
Demo

Run VIS

```
barbara@mtcpc3> vis
vis release 2.1 (compiled 8-Nov-09 at 2:01 AM)
vis> read_verilog arbiter.v
arbiter.v
Warning: Model arbiter may have a cyclic connection which involves variable ack<0>
vis> init_verify
vis> model_check arbiter.ctl
# MC: formula passed --- AG(!((ack<0>=1 * ack<1>=1)))
# MC: formula passed --- AG(AF((lreq<0>=1 -> ack<0>=1)))
# MC: formula passed --- AG(AF((lreq<1>=1 -> ack<1>=1)))
# MC: formula passed --- AG((ack<0>=1 -> lreq<0>=1))
# MC: formula passed --- AG((ack<1>=1 -> lreq<1>=1))
vis>
```
Cabbage, Goat, Boat
typedef enum {NONE,CABBAGE,GOAT,WOLF} pType;

// Slide: left=0, right=1
module puzzel(clock, select, boat, cabbage, goat, wolf);
  input  clock, select;
  output boat, cabbage, goat, wolf;
pType wire select;
  reg boat, cabbage, goat, wolf;
initial begin
  boat = 0;
  cabbage = 0;
  goat = 0;
  wolf = 0;
end
always @(posedge clock) begin
  if ((select==CABBAGE) && (cabbage == boat)) cabbage = ~cabbage;
  if ((select==GOAT) && (goat == boat))    goat = ~goat;
  if ((select==WOLF) && (wolf == boat))    wolf = ~wolf;
  boat=~boat;
end
endmodule
barbara@mtcpc3> vis
vis release 2.1 (compiled 8-Nov-09 at 2:01 AM)
vis> read_verilog puzzel.v
puzzel.v
vis> init_verify
vis> simulate -n 6
# vis release 2.1 (compiled 8-Nov-09 at 2:01 AM)
# Network: puzzel
# Simulation vectors have been randomly generated

.inputs select
.latches boat cabbage goat wolf
.outputs boat cabbage goat wolf
.initial 0 0 0 0

.start_vectors

# select ; boat cabbage goat wolf ; boat cabbage goat wolf
CABBAGE ; 0 0 0 0 ; 0 0 0 0
GOAT ; 1 1 0 0 ; 1 1 0 0
CABBAGE ; 0 1 1 0 ; 0 1 1 0
NONE ; 1 1 1 0 ; 1 1 1 0
WOLF ; 0 1 1 0 ; 0 1 1 0
NONE ; 1 1 1 0 ; 1 1 1 0
Goal

- Cabbage, goat, and wolf are on the other side
  - $\text{EF}(\text{cabbage}=1 \times \text{goat}=1 \times \text{wolf}=1)$;
  - Yes
  - $\text{AG}(! (\text{cabbage}=1 \times \text{goat}=1 \times \text{wolf}=1))$;
  - No + counterexample
    - `model_check -d 1 -i puzzel.ctl`
Extend with Eating

- Goat eats cabbage if farmer is away
- Wolf eats goat if farmer is away

```verilog
module ...

reg eating;

initial begin
    eating=0;
end

always @(posedge clock) begin
    if ((cabbage==goat) && ~(goat==boat))
        eating=1;
    if ((goat==wolf) && ~(wolf==boat))
        eating=1;
end
endmodule
```
Goal

• Cabbage, goat, and wolf are on other side without been eating before

- \( E( (eating=0) U (cabbage=1 \ast \text{goat}=1 \ast \text{wolf}=1 \ast \text{eating}=0)) \);
- Yes
- \( \neg E( (eating=0) U (cabbage=1 \ast \text{goat}=1 \ast \text{wolf}=1 \ast \text{eating}=0)) \);
- No + counterexample
Computational Tree Logic

- CTL [CES86] is a temporal logic – it talks about time
- Time is used to refer to steps in a computation (e.g., holds initially, after two step, always/globally)
- Tree: it refers to “computation tree”
• Recall, 2-bit counter
CTL: Syntax

• CTL is defined over set A of atomic propositions called alphabet
  – E.g., ack[0]=1, cabbage=1
• Any atomic proposition is a formula
• Given two CTL formulas \( \varphi \) and \( \psi \), then
  – \( \neg \varphi \), \( \varphi \land \psi \), \( \varphi \lor \psi \) are CTL formulas and
  – EX\( \varphi \), E\( \varphi \lor \psi \), and EG\( \varphi \) are CTL formulas.
  – Note that AG, AX,... can be defined with operators above (we'll see later)
CTL: Syntax

• CTL operators are composed of
  1. a path quantifier:
     E – there exists a path
  2. a temporal operator:
     X – next,
     U – until,
     G – henceforth/globally
CTL: Semantics

• CTL semantics is defined over Kripke structures K=(S,T,s₀,A,L)
  – S...finite set of states
  – T ⊆ S×S is the transition relation
    • We assume that every state has at least one successor
  – s₀ ∈ S...is a initial state
  – A...set of atomic propositions
  – L: S → 2^A is a labeling function stating with propositions hold at each state
Example: Kripke Structure

\[ S = \{a, b, c\} \]
\[ T = \{(a, b), (b, a), (b, c), (c, c), (c, a)\} \]
\[ s_0 = a \]
\[ A = \{p, q\} \]
\[ L(a) = \{\} \]
\[ L(b) = \{p\} \]
\[ L(c) = \{q\} \]
Recall: 2-Bit Counter
Recall: 2-Bit Counter

S = \{a, b, c, d\}
T = \{(a, a), (a, b), (b, b), (b, c), (c, c), (c, d), (d, d), (d, a)\}

s_0 = a
A = \{m\}
L(a) = L(b) = L(c) = \{\}
L(d) = \{m\}
CTL: Semantics

- Given Kripke structure $K$, we say state $s$ of $K$ models $\varphi$ written as $K,s \models \varphi$ if the satisfaction relation defined below holds ($\models$ is called double turnstile).
- Satisfaction of a CTL formula is defined inductively as follows:
  - $K,s \models \varphi$ iff $\varphi \in L(s)$ for $\varphi \in A$
  - $K,s \models \neg \varphi$ iff $s \not\models \varphi$
  - $K,s \models \varphi \lor \psi$ iff $s \models \varphi$ or $s \models \psi$
CTL: Semantics

- \( K,s \models \varphi \land \psi \) iff \( s \models \varphi \) and \( s \models \psi \)
- \( K,s \models \text{EX } \varphi \) iff there exists a path \( s_0,s_1,... \) in \( K \) such that \( s=s_0 \) and \( s_1 \models \varphi \)
- \( K,s \models \text{EG } \varphi \) iff there exists a path \( s_0,s_1,... \) in \( K \) such that \( s=s_0 \) and for all \( i \geq 0 \), \( K,s_i \models \varphi \) holds
- \( K,s \models \text{E}\varphi \text{U}\psi \) iff there exists a path \( s_0,s_1,... \) in \( K \) such that \( s=s_0 \) and there exists \( i \geq 0 \) for which \( K,s_i \models \psi \) and for all \( 0 \leq j < i \), \( K,s_j \models \varphi \) holds
Examples

- First four cases are elementary
- Examples for the last three cases
CTL: Semantics

• A formula $\varphi$ holds in a Kripke structure, written $K \models \varphi$, if it is satisfied in the initial state of $K$.
• Checking if $K \models \varphi$ is called CTL Model Checking.
Example

Does it satisfy $\text{EG} \neg p$?
Example

Does it satisfy $\text{EG} \neg p$?
We need an infinite path starting at state $a$ along which $p$ is always false.
Abbreviations

\( \varphi \rightarrow \psi \) (implies) ... \\
\( \varphi \oplus \psi \) (exclusive or) ... \\
\( \varphi \leftrightarrow \psi \) (equivalence) ... \\
EF \varphi ... \\
AX \varphi ... \\
AG \varphi ... \\
AF \varphi ... \\
A \varphi U \psi ... \\
A \varphi R \psi \) (releases) \( \neg (E \neg \varphi U \neg \psi) \)

E \varphi R \psi \) (releases) \( \neg (A \neg \varphi U \neg \psi) \\

**Note:**
true = \( \varphi \lor \neg \varphi \) 
false = \( \varphi \land \neg \varphi \) 
EF_\varphi = E true U_\varphi 
EG_\varphi = E false R_\varphi 

s \models E \varphi R \psi \) if there exists a path starting at s along which either \( \varphi \) holds forever, or \( \varphi \) holds up to a state where both \( \varphi \) and \( \psi \) hold.
Abbreviations

\( \varphi \rightarrow \psi \) (implies)
\( \neg \varphi \lor \psi \)

\( \varphi \oplus \psi \) (exclusive or)
\( (\varphi \land \neg \psi) \lor (\neg \varphi \land \psi) \)

\( \varphi \leftrightarrow \psi \) (equivalence)
\( (\varphi \land \psi) \lor (\neg \varphi \land \neg \psi) \)

EF \( \varphi \)
E(\( \varphi \lor \neg \varphi \lor \varphi \))

AX \( \varphi \)
\( \neg \text{EX} \neg \varphi \)

AG \( \varphi \)
\( \neg \text{EF} \neg \varphi \)

AF \( \varphi \)
\( \neg \text{EG} \neg \varphi \)

A\( \varphi \lor \psi \)
\( \neg (\text{E} \neg \varphi \lor (\varphi \lor \psi)) \land \neg \text{EG} \neg \varphi \)

A\( \varphi \rightarrow \psi \) (releases)
\( \neg (\text{E} \neg \varphi \lor \neg \psi) \)

E\( \varphi \rightarrow \psi \) (releases)
\( \neg (\text{A} \neg \varphi \lor \neg \psi) \)

Note:
true = \( \varphi \lor \neg \varphi \)
false = \( \varphi \land \neg \varphi \)
EF\( \varphi \) = E true U\( \varphi \)
EG\( \varphi \) = E false R\( \varphi \)
• So far, future-tense CTL
• Past-tense CTL
  – $EX\varphi \rightarrow EY\varphi$, there exists a predecessor
  – $EG\varphi \rightarrow EH\varphi$, hitherto (always up-to-now)
  – $E\varphi U\psi \rightarrow E\varphi S\psi$, since
  – $EF\varphi \rightarrow EP\varphi$, previously
• We will use only future-tense CTL
Tell me some formulas that hold and that do not hold!
True or false?
AX(p=1)
EG(q=1)
EF(EG(q=1))
AG(q=0)
EF(q=1)
E(p=1 U q=1)
AX EX EG(q=1)
EX E(p=1 U q=1)
EG(q=0)

Tell me some formulas that hold and that do not hold!