Embedded Systems 10

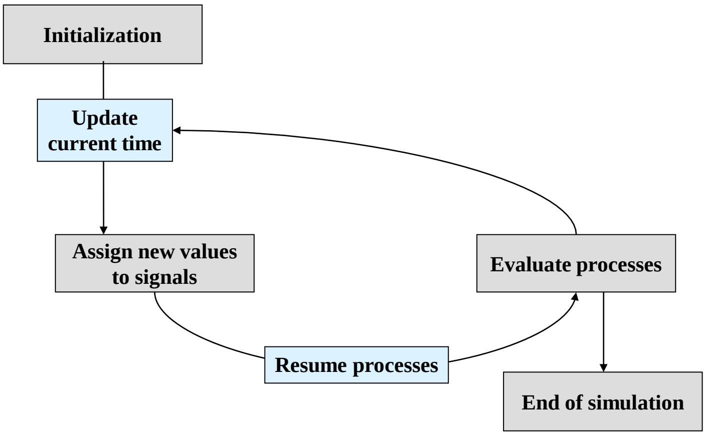
Midterm exam Thursday May 31, 16:00-18:00

- **Groups 1 – 12:** HS 002
- **All other groups:** Günter Hotz lecture hall

- The exam will be **open book**. That is, you are allowed to use printouts of the lecture slides, books, and any handwritten notes during the exam.

- **No discussion slots next week**
- Last two problem sets will be discussed in **Tutorial on Wednesday**

REVIEW: Overview of simulation



REVIEW: Transaction list and process activation list

- Transaction list
  - For signal assignments
  - Entries of form (*s*, *v*, *t*) meaning „signal *s* is set to value *v* at time *t*“
  - Example: (clock, ´1´, 10 ns)
- Process activation list
  - For reactivating processes
  - Entries of form (p_j, *t*) meaning „process p_i resumes at time *t*“.

REVIEW: Initialization

- At the beginning of initialization, the current time, t_{curr}, is assumed to be 0 ns.
- An initial value is assigned to each signal.
  - Taken from declaration, if specified there, e.g.,
    - **signal** s : std_ulogic := `0`;
  - Otherwise: First value in enumeration for enumeration based data types, e.g.
    - **signal** s : std_ulogic
      with
      type std_ulogic **is** (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`);
      initial value is `U`
  - This value is assumed to have been the value of the signal for an infinite length of time prior to the start of the simulation.
- Initialization phase executes each process exactly once (until it suspends).
- During execution of processes: Signal assignments are collected in transaction list (**not** executed immediately!) – more details later.
- If process stops at „wait for“-statement, then update process activation list – more details later.
- After initialization the time of the next simulation cycle (which in this case is the first simulation cycle), t_{next} is calculated:
  - Time t_{next} of the next simulation cycle = earliest of
    1. time'high (end of simulation time).
    2. Earliest time in transaction list (if not empty)
    3. Earliest time in process activation list (if not empty).

REVIEW: Signal assignment phase – first part of step

- Each simulation cycle starts with setting the current time to the next time at which changes must be considered:
- $t_{curr} = t_{next}$
- This time t_{next} was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time would exceed its maximum, time’high.
- For all (s, v, t_{curr}) in transaction list:
  - Remove (s, v, t_{curr}) from transaction list.
  - s is set to v.
- For all processes p_i which wait on signal s:
  - Insert (p_i, t_{curr}) in process activation list.
- Similarly, if condition of „**wait until**“-expression changes value.

REVIEW: Process execution phase – second part of step (1)

- Resume all processes p_i with entries (p_i, t_{curr}) in process activation list.
- Execute all activated processes „in parallel“ (in fact: in arbitrary order).
- Signal assignments
  - are collected in transaction list (**not** executed immediately!).
  - Examples:
    - *s* <= *a* **and** *b*;
      - Let *v* be the conjunction of current value of *a* and current value of *b*.
      - Insert (*s, v,* t_{curr}) in transaction list.
    - *s* <= ´1´ **after** 10 ns;
      - Insert (*s*, ´1´, t_{curr} + 10 ns) into transaction list.
- Processes are executed until wait statement is encountered.
- If process p_i stops at „wait for“-statement, then update process activation list:
  - Example:
    - p_i stops at „**wait for** 20 ns;“
    - Insert (p_i, t_{curr} + 20 ns) into process activation list

REVIEW: Process execution phase – second part of step (2)

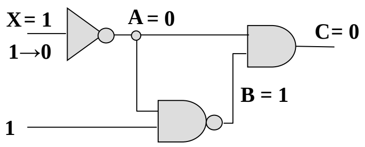
If some process reaches last statement and

- does not have a sensitivity list and
- last statement is not a wait statement,

then it continues with first statement and runs until wait statement is reached.

- When all processes have stopped, the time of the next simulation cycle t_{next} is calculated:
  - Time t_{next} of the next simulation cycle = earliest of
    1. time’high (end of simulation time).
    2. Earliest time in transaction list (if not empty)
    3. Earliest time in process activation list (if not empty).
- Stop if t_{next} = time’high and transaction list and process activation list are empty.

REVIEW: Delta delay



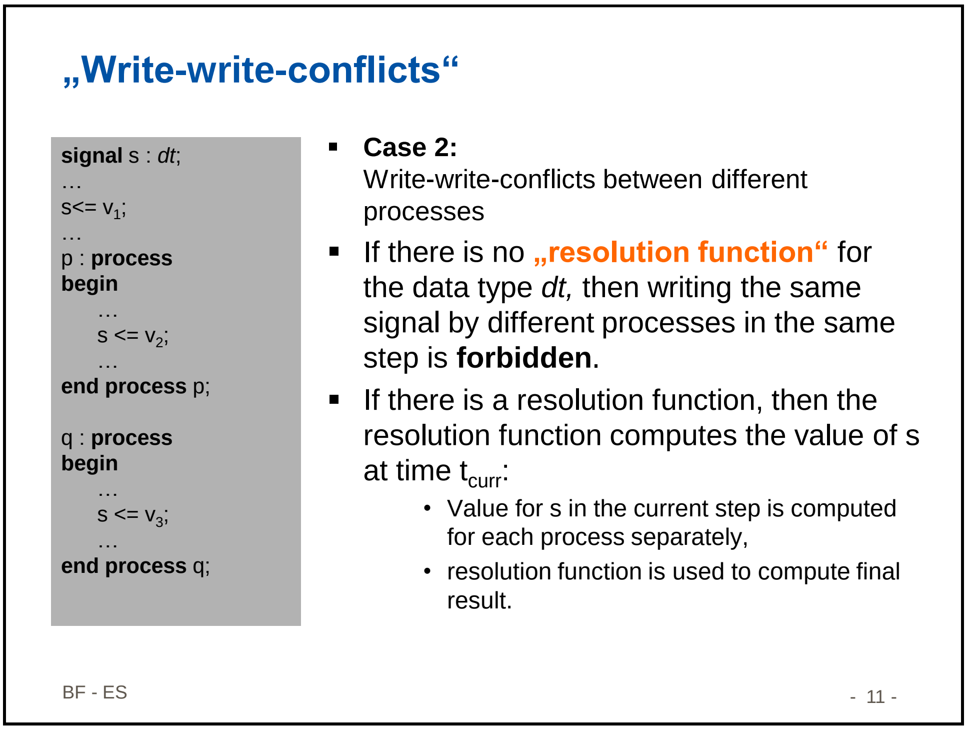
Simulation time does not proceed due to delta delays!

Current time	Delta delay	Event
0 ns	1	-- evaluation of inverter -- (A, 1, 0 ns)
	2	-- evaluation of AND and NAND -- (B, 0, 0ns), (C, 1, 0ns)
	3	-- evaluation of AND -- (C, 0, 0ns)

„Write-write-conflicts“

```
signal s : bit;
…
p : process
begin
      …
      s <= `0`;
      …
      s <= `1`;
      wait for 5 ns;
end process p;
```

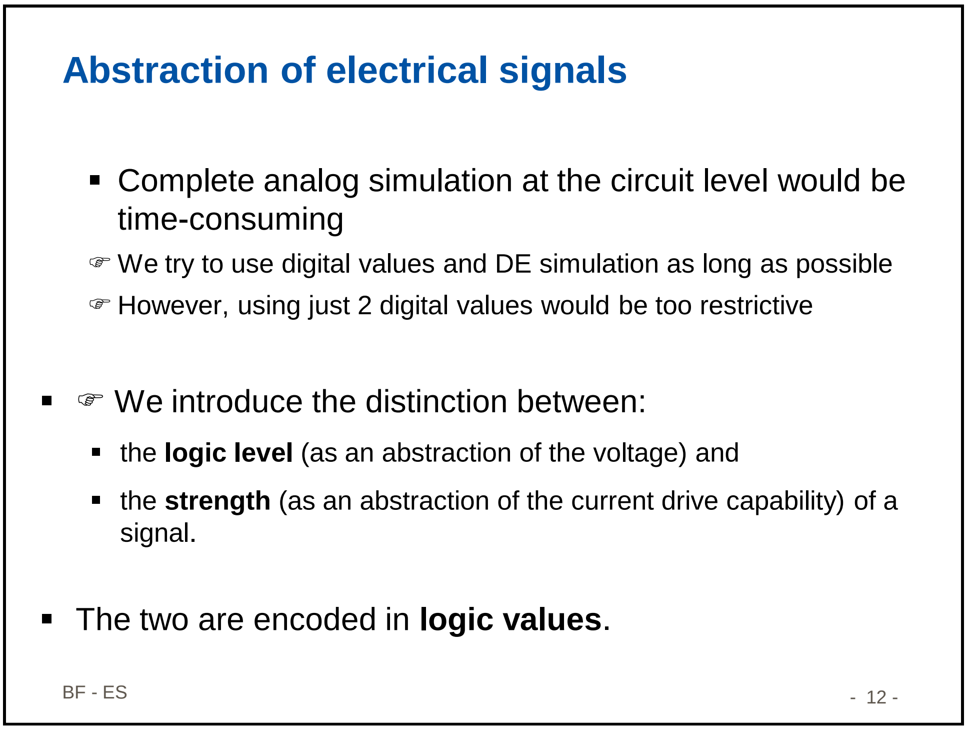
- **Case 1:**
  Write-write-conflicts are restricted to the same process
  (i.e. they occur inside the same process)
  - Then the second signal assignment overwrites the first one.
  - This is the only case of „non-concurrency“ of signal assignments
  - Note that writing to different signals occurs concurrently, however!



„Write-write-conflicts“

```
signal s : dt;
…
s<= v1;
…
p : process
begin
   …
   s <= v2;
   …
end process p;

q : process
begin
   …
   s <= v3;
   …
end process q;
```

- **Case 2:**
  Write-write-conflicts between different processes
- If there is no **„resolution function“** for the data type *dt,* then writing the same signal by different processes in the same step is **forbidden**.
- If there is a resolution function, then the resolution function computes the value of s at time t_{curr}:
  - Value for s in the current step is computed for each process separately,
  - resolution function is used to compute final result.



Abstraction of electrical signals

- Complete analog simulation at the circuit level would be time-consuming

☞ We try to use digital values and DE simulation as long as possible

☞ However, using just 2 digital values would be too restrictive

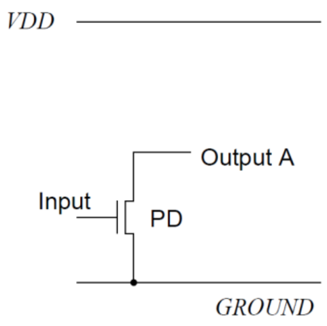
- ☞ We introduce the distinction between:
  - the **logic level** (as an abstraction of the voltage) and
  - the **strength** (as an abstraction of the current drive capability) of a signal.
- The two are encoded in **logic values**.

1 signal strength

- Logic values '0' and '1'.
- Both of the same strength.
- Encoding false and true, respectively.

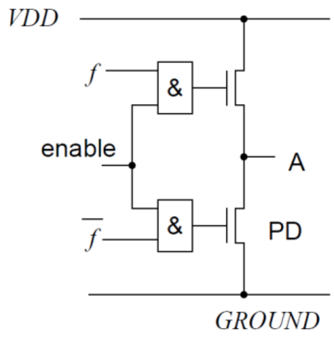
2 signal strengths

- Many subcircuits can effectively disconnect themselves from the rest of the circuit (they provide “high impedance“ values to the rest of the circuit).
- Example: subcircuits with open collector



Input = ’0’ –> A disconnected

TriState circuits

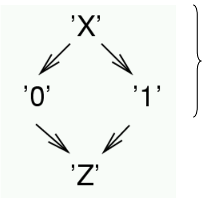


enable = ’0’ –> A disconnected

☞ We introduce signal value 'Z', meaning “high impedance“

2 signal strengths (cont’ed)

- We introduce an operation #, which generates the effective signal value whenever two signals are connected by a wire.
- #('0','Z')='0'; #('1','Z')='1'; '0' and '1' are “stronger“ than 'Z'



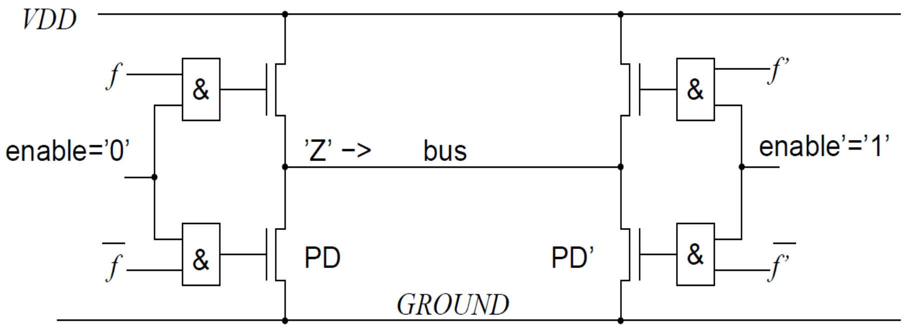
1 strength

Hasse diagram

According to the partial order in the diagram, # returns the smallest element at least as large as the *two arguments* (“Sup”).

In order to define #('0','1'), we introduce 'X', denoting an undefined signal level.
'X' has the same strength as '0' and '1'.

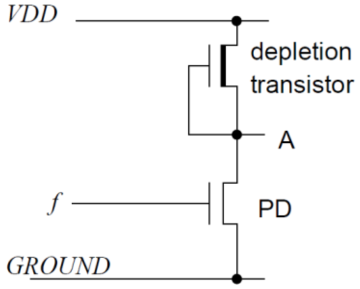
Application example



signal value on bus = #(value from left subcircuit, value from right subcircuit)

#('Z', value from right subcircuit) = value from right subcircuit
“as if left circuit were not there“.

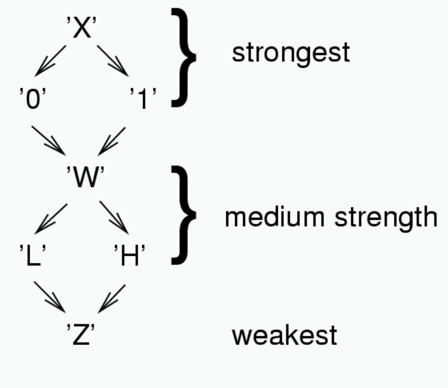
3 signal strengths



Depletion transistor contributes a weak value to be considered in the #-operation for signal A
☞ Introduction of 'H',
denoting a weak signal of the same level as '1'.
#('H', '0')='0'; #('H','Z') = 'H'

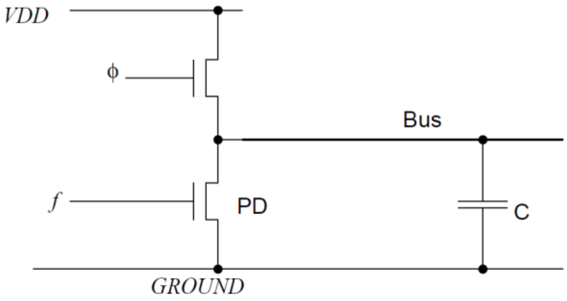
3 signal strengths

- There may also be weak signals of the same level as '0'
- ☞ Introduction of 'L', denoting a weak signal of the same level as '0': #('L', '1')=‘1'; #('L','Z') = 'L';
- ☞ Introduction of 'W', denoting a weak signal of undefined level 'X': #('L', 'H')='W'; #('L','W') = 'W';
- # reflected by the partial order shown.



4 signal strengths (1)

- pre-charging:



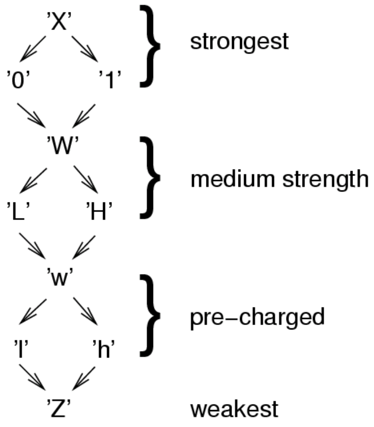
Pre-charged '1'-levels weaker than any of the values considered so far, except 'Z'.

☞ Introduction of 'h', denoting a very weak signal of the same level as '1'.

#('h', '0')='0'; #('h','Z') = 'h'

4 signal strengths (2)

- There may also be weak signals of the same level as '0'
- ☞ Introduction of 'l', denoting a very weak signal of the same level as '0': #('l', '0')='0'; #('l,'Z') = 'l';
- ☞ Introduction of 'w', denoting a very weak signal of the same level as 'W': #('l', 'h')='w'; #('h','w') = 'w'; ...
- # reflected by the partial order shown.



IEEE 1164

- VHDL allows user-defined value sets.

→ Each model could use different value sets (unpractical)

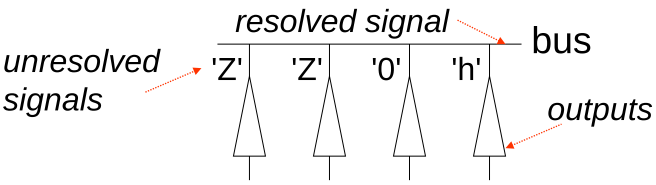
→ Definition of standard value set according to standard IEEE 1164:

{'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}

- First seven values as discussed previously.
- 'U': un-initialized signal; used by simulator to initialize all not explicitly initialized signals:
  type std_ulogic **is** (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`);
- '-': is used to specify don’t cares:
  - Example: **if** a /= ‘1’ **or** b/=‘1’ **then** f <= a **exor** b; **else** f <= ‘-’;
  - ‘-’ may be replaced by arbitrary value by synthesis tools.

Outputs tied together

In hardware, connected outputs can be used:



resolved signal

bus

unresolved signals

'Z' 'Z' '0' 'h'

outputs

Modeling in VHDL: resolution functions

type std_ulogic **is** ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

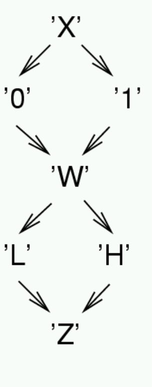
subtype std_logic **is** resolved std_ulogic;

Resolution function for IEEE 1164

```
type std_ulogic_vector is array(natural range<>)of std_ulogic;

function resolved (s:std_ulogic_vector) return std_logic is
  variable result: std_ulogic:='Z';   --weakest value is default
  begin
    if (s'length=1) then return s(s'low) --no resolution
    else for i in s'range loop
      result:=resolution_table(result,s(i))
    end loop
    end if;
   return result;
  end resolved;
```

Resolution function for IEEE 1164

```
constant resolution_table : stdlogic_table := (
--U    X    0    1    Z   W     L    H   –
('U',  'U',  'U',  'U', 'U', 'U',   'U', 'U', 'U'),   --| U |
('U',  'X',  'X',  'X', 'X', 'X',   'X', 'X', 'X'),   --| X |
('U',  'X',  '0',  'X', '0', '0',   '0', '0', 'X'),   --| 0 |
('U',  'X',  'X',  '1', '1', '1',   '1', '1', 'X'),   --| 1 |
('U',  'X',  '0',  '1', 'Z', 'W',   'L', 'H', 'X'),   --| Z |
('U',  'X',  '0',  '1', 'W', 'W',   'W', 'H', 'X'),   --| W |
('U',  'X',  '0',  '1', 'L', 'W',   'L', 'W', 'X'),   --| L |
('U',  'X',  '0',  '1', 'H', 'W',   'W', 'H', 'X'),   --| H |
('U',  'X',  'X',  'X', 'X', 'X',   'X', 'X', 'X')    --| - |
);
```



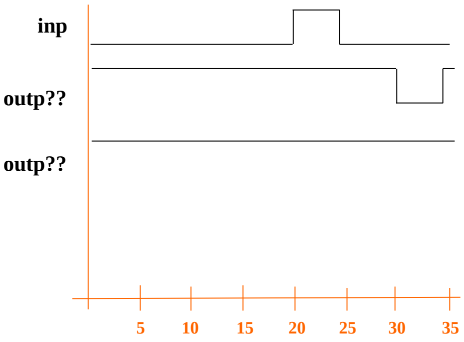
Inertial and transport delay model

- Signal assignment:
  - signal_assignment ::=
    target <= [delay_mechanism] waveform_element
    { , waveform_element }
  - waveform_element ::=
    value_expression [**after** time_expression]
  - delay_mechanism ::=
    transport | [**reject** time_expression] inertial
- Example:
  - Inpsig <= ´0´, ´1´after 5 ns, ´0´ after 10 ns, ´1´ after 20 ns;

Inertial and transport delay model

- Example for signal assignment:

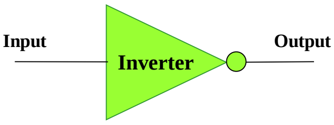
  outp <= **not** inp **after** 10 ns;



Inertial and transport delay model

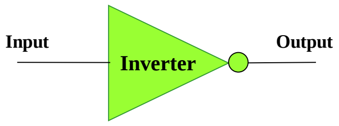
Two delay models in VHDL:

- **Inertial delay** („*träge Verzögerung*“)
- **Transport delay** („*nichtträge Verzögerung*“)



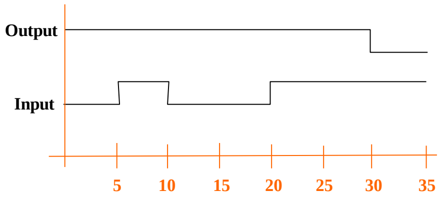
- Inertial delay model is motivated by the fact that physical gates absorb short pulses (spikes) at their inputs (due to internal capacities)

Inertial delay model



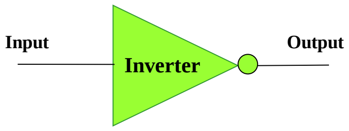
- … is the **default** model
- Absorbs pulses at the inputs which are shorter than the delay specified for the gate / operation

```
-- INERTIAL is the default
Output <= NOT input AFTER 10 ns;
```

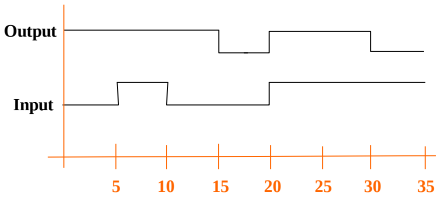


Transport delay model

- Transmits all pulses at the inputs ideally



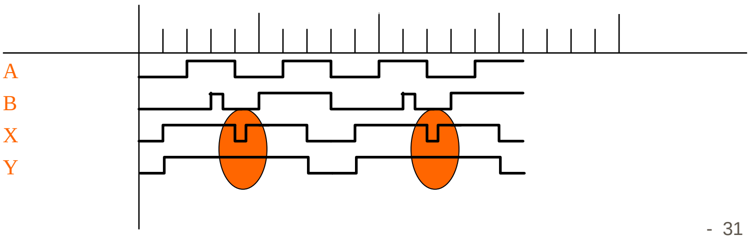
```
-- TRANSPORT must be specified
Output <= TRANSPORT NOT input AFTER 10 ns;
```



Inertial and transport delay model

```
entity DELAY is
end DELAY;

architecture RTL of DELAY is
  signal A, B, X, Y: bit;
begin
  p0: process (A, B)
  begin
    Y <= A nand B after 10 ns;
    X <= transport A nand B after 10 ns;
  end process;
```

```
  p1: process
  begin
    A <= '0', '1' after 20 ns, '0'
              after 40 ns, '1' after 60 ns;
    B <= '0', '1' after 30 ns, '0'
              after 35 ns, '1' after 50 ns;
    wait for 80 ns;
  end process
end RTL;
```



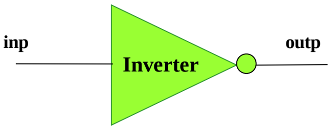
Semantics of transport delay model

Signal assignments change transaction list.

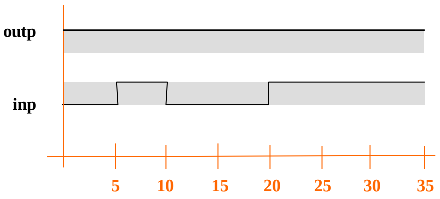
- Before transaction (s, t_1, v_1) is inserted into transaction list, all transactions in the transaction list (s, t_2, v_2) with $t_2 \geq t_1$ are removed from transaction list.

Example for transport delay model

```
inv : process(inp)
begin
    if inp=`1` then
            outp <= transport `0` after 20 ns;
    elsif inp=`0` then
            outp <= transport `1` after 12.5 ns
    end if;
end process inv;
```



- Transaction list:
    - At 5ns:
      (outp, 25ns, `0`)
    - At 10 ns:
      (outp, 22.5ns, `1`), (outp, 25ns, `0`)
      Remove (outp, 25ns, `0`)!
      → (outp, 22.5ns, `1`)



Semantics of inertial delay model

- Semantics for more general version of inertial delay statement:
    - Inertial delay absorbs pulses at the inputs which are shorter than the delay specified for the gate / operation.
    - Key word **reject** permits absorbing only pulses which are shorter than specified delay:
        - Example:
            - outp <= **reject** 3 ns **inertial not** inp **after** 10 ns;
            - Only pulses smaller than 3 ns are absorbed.
            - outp <= **reject** 10 ns **inertial not** inp **after** 10 ns;
              and
              outp <= **not** inp **after** 10 ns;
              are equivalent.

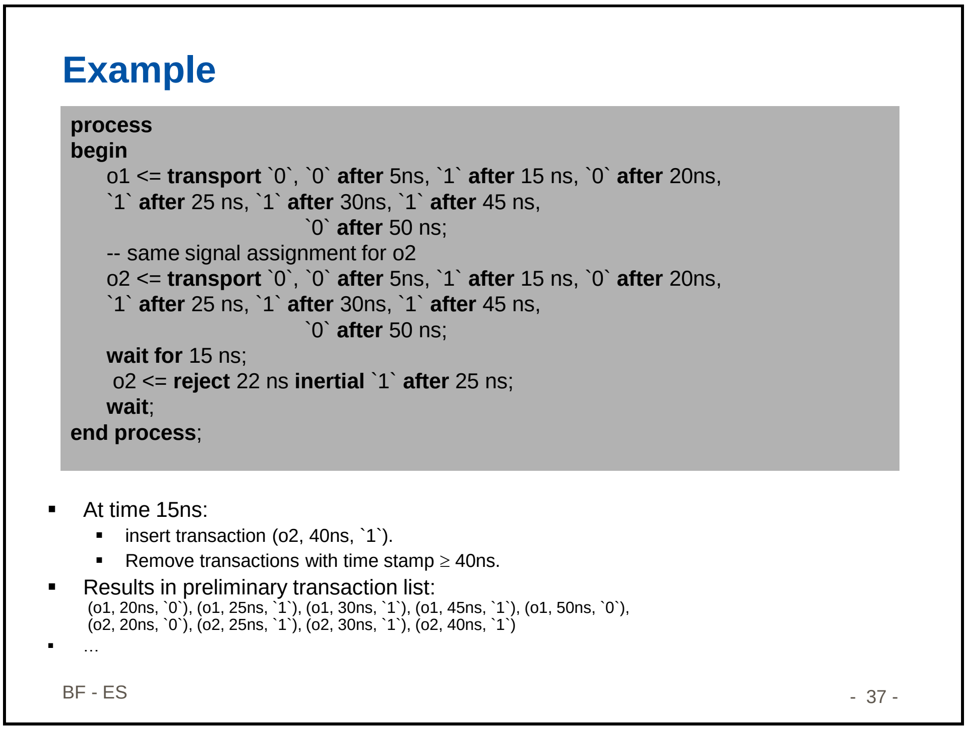
Semantics of inertial delay model

- Rule 1 as for transport delay model:
  Before transaction (s, t_1, v_1) is inserted into transaction list, all transactions in the transaction list (s, t_2, v_2) with $t_2 \geq t_1$ are removed from transaction list.
- Rule 2 removes also some transactions with times < t_1:
  - Suppose the time limit for reject is rt.
  - Transactions for signal s with time stamp in the intervall (t_1 – rt, t_1) are removed.
  - Exception:
    If there is in (t_1 – rt, t_1) a subsequence of transactions for s immediately before (s, t_1, v_1) which also assign value v_1 to s, then these transactions are preserved.

Example

```
process
begin
    o1 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
    `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                          `0` after 50 ns;
    -- same signal assignment for o2
    o2 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
    `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                          `0` after 50 ns;
    wait for 15 ns;
     o2 <= reject 22 ns inertial `1` after 25 ns;
    wait;
end process;
```

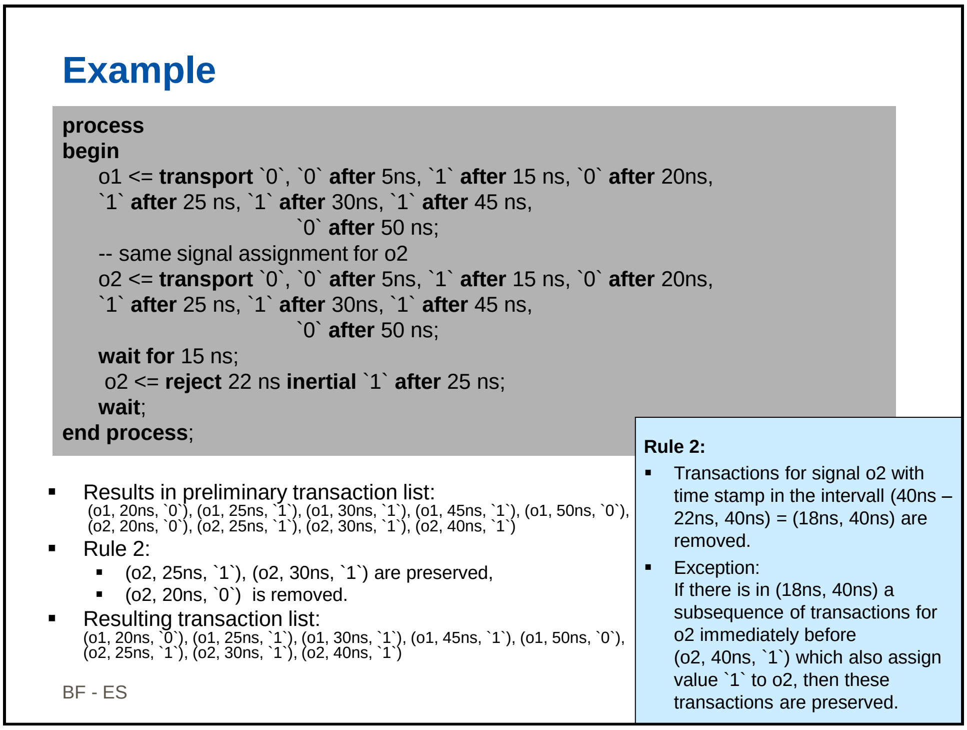
- Transaction list until „**wait for** 15 ns“:
  (o1, 0ns, `0`), (o1, 5ns, `0`), (o1, 15ns, `1`), (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 0ns, `0`), (o2, 5ns, `0`), (o2, 15ns, `1`), (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 45ns, `1`), (o2, 50ns, `0`)
- Transaction list when process is reactivated at time 15ns:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 45ns, `1`), (o2, 50ns, `0`)
- …



Example

```
process
begin
    o1 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
    `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                      `0` after 50 ns;
    -- same signal assignment for o2
    o2 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
    `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                      `0` after 50 ns;
    wait for 15 ns;
     o2 <= reject 22 ns inertial `1` after 25 ns;
    wait;
end process;
```

- At time 15ns:
  - insert transaction (o2, 40ns, `1`).
  - Remove transactions with time stamp ≥ 40ns.
- Results in preliminary transaction list:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 40ns, `1`)
- …



Example

```
process
begin
    o1 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
    `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                      `0` after 50 ns;
    -- same signal assignment for o2
    o2 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
    `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                      `0` after 50 ns;
    wait for 15 ns;
     o2 <= reject 22 ns inertial `1` after 25 ns;
    wait;
end process;
```

- Results in preliminary transaction list:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 40ns, `1`)
- Rule 2:
  - (o2, 25ns, `1`), (o2, 30ns, `1`) are preserved,
  - (o2, 20ns, `0`) is removed.
- Resulting transaction list:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 40ns, `1`)

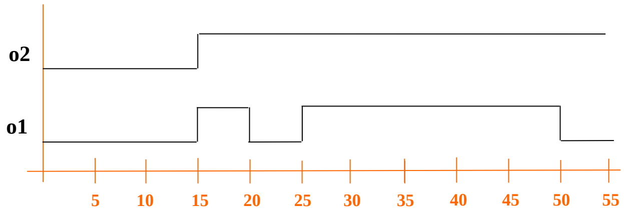
Rule 2:

- Transactions for signal o2 with time stamp in the intervall (40ns – 22ns, 40ns) = (18ns, 40ns) are removed.
- Exception:
  If there is in (18ns, 40ns) a subsequence of transactions for o2 immediately before (o2, 40ns, `1`) which also assign value `1` to o2, then these transactions are preserved.

Example

```
process
begin
     o1 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
     `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                              `0` after 50 ns;
     -- same signal assignment for o2
     o2 <= transport `0`, `0` after 5ns, `1` after 15 ns, `0` after 20ns,
     `1` after 25 ns, `1` after 30ns, `1` after 45 ns,
                              `0` after 50 ns;
     wait for 15 ns;
      o2 <= reject 22 ns inertial `1` after 25 ns;
     wait;
end process;
```

- Resulting wave form:



Functions and procedures

- Apart from entities / architectures there are also functions and procedures in the usual (software) sense.
- Functions are typically used for providing conversion between data types or for defining operators on user-defined data types.
- Procedures may have parameters of directions **in**, **out** and **inout**.
  - **in** comparable to *call by value*,
  - **out** for providing results,
  - **inout** comparable to *call by reference*.

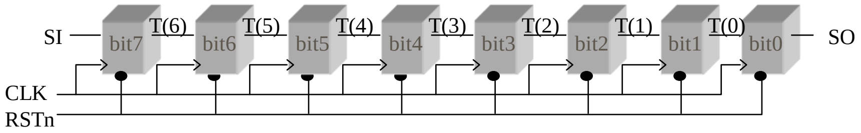
Example

```
architecture RTL of TEST is
  function BOOL2BIT (BOOL: boolean) return bit is
  begin
    if BOOL then return '1'; else return '0'; end if;
  end BOOL2BIT;

  procedure EVEN_PARITY (
      signal D: in bit_vector(7 downto 0);
      signal PARITY : out bit ) is
    variable temp : bit;
  begin
    ....
  end;

  signal DIN : bit_vector(7 downto 0);
  signal BOOL1 : boolean;
  signal BIT1, PARITY : bit;
 begin
  do_it: process (BOOL1, DIN)
  begin
    BIT1 <= BOOL2BIT(BOOL1);
    EVEN_PARITY(DIN, PARITY);
  end process;
  ....
 end;
```

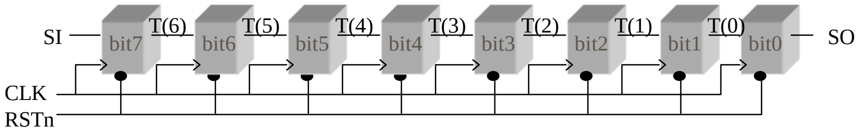
Parameterized hardware

- Conditional component instantiation with **if … generate** construct.
- Iterative component instantiation with **for … generate** construct.
- Parameterized design with **generic** parameters.

Example: 8-bit shift register



```
entity SHIFT8 is
port ( RSTn, CLK, SI : in std_logic;
       SO : out std_logic );
end SHIFT8;
```



```
architecture RTL1 of SHIFT8 is

  component DFF
  port ( RSTn, CLK, D: in std_logic;
         Q           : out std_logic );
  end component;
  signal T: std_logic_vector(6 downto 0);

begin

  bit7 : DFF
      port map (RSTn => RSTn, CLK => CLK,
                D => SI, Q => T(6) );
  bit6 : DFF
      port map (RSTn => RSTn, CLK => CLK,
                D => T(6), Q => T(5) );
  bit5 : DFF
      port map (RSTn, CLK, T(5), T(4) );
  ...
  bit1 : DFF
      port map (RSTn, CLK, T(1), T(0) );
  bit0 : DFF
      port map (RSTn, CLK, T(0), S0 );

end RTL1;
```

Example: 1024-bit shift register

```
architecture RTL2 of SHIFT1024 is
  component DFF
  port ( RSTn, CLK, D: in std_logic;
         Q           : out std_logic );
  end component;
  signal T: std_logic_vector(1022 downto 0);
begin
  g0: for i in 1023 downto 0 generate
      g1: if (i = 1023) generate
            bit1023 : DFF port map (RSTn,CLK,SI,T(1022));
          end generate;
      g2: if (i>0) and (i<1023) generate
            bitm : DFF port map (RSTn,CLK,T(i),T(i-1));
          end generate;
      g3: if (i=0) generate
            bit0 : DFF port map (RSTn,CLK,T(0),S0);
          end generate;
    end generate;

end RTL2;
```

Example: n-bit shift register

```
entity SHIFTn is
generic ( n : positive);
port ( RSTn, CLK, SI : in std_logic;
       SO : out std_logic );
end SHIFTn;
```

```
architecture RTL3 of SHIFTn is
  component DFF
  port ( RSTn, CLK, D: in std_logic;
         Q           : out std_logic );
  end component;
  signal T: std_logic_vector(n-2 downto 0);
begin
  g0: for i in n-1 downto 0 generate
      g1: if (i = n-1) generate
            bit_high : DFF port map (RSTn,CLK,SI,T(n-2));
          end generate;
      g2: if (i>0) and (i<n-1) generate
            bitm : DFF port map (RSTn,CLK,T(i),T(i-1));
          end generate;
      g3: if (i=0) generate
            bit0 : DFF port map (RSTn,CLK,T(0),S0);
          end generate;
    end generate;
end RTL3;
```

Example: n-bit shift register

- Component instantiation

```
…
  component SHIFTn is
  generic ( n : positive);
  port ( RSTn, CLK, SI : in std_logic;
          SO : out std_logic );
  end component;
```

```
…
begin
  …
  Shift32comp : SHIFTn
    generic map (n => 32)
    port map(RSTn => …,
                  CLK => …,
                  SI => …,
                  SO => …);
  …
end;
```

VHDL: Evaluation

- Hierarchical specification by entities / architectures / components, (procedures and functions)
- no nested processes
- Static number of processes
- Complicated simulation semantics
- May be too low level for initial, abstract specification of very large systems
- Mainly used for hardware simulation+synthesis

REVIEW: computational models

Communication/ local computations	Shared memory	Asynchronous message passing
Communicating finite state machines	Statecharts, hybrid automata, synchronous composition	
Data flow		Petri nets, Kahn process networks, SDF
Discrete event (DE) model	Simulink, VHDL	Distributed DE

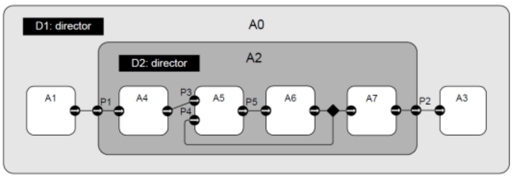
Combinations of computational models

Ptolemy

- Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation.

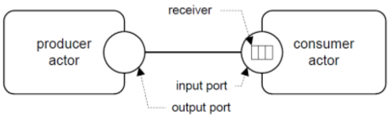
  http://ptolemy.berkeley.edu/

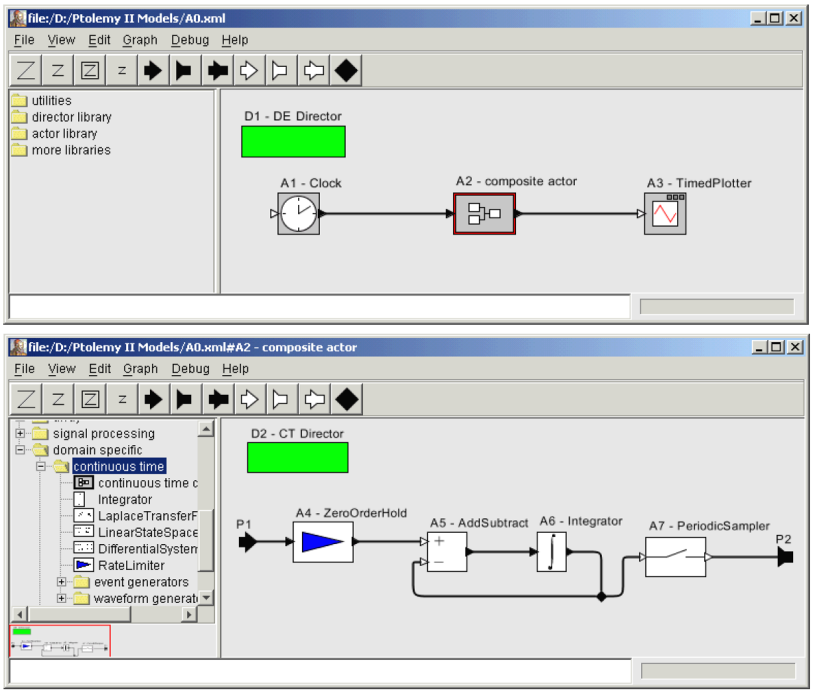
  - discrete-event systems
  - SDF
  - process networks
  - Petri nets
  - priority-based schedules
  - synchronous/reactive
  - Finite-state machines
  - continuous-time
  - modal systems
  - Graphics, 3D animations



Ptolemy

- A model is a set of interconnected *actors* and one *director*
- Actor
  - Input & output *ports*, states, & parameters
- Models of computation
  - Define the interaction semantics
  - Implemented in Ptolemy II by a *domain*
    - Director + Receiver
- Director
  - Manages the data flow and the scheduling of the actors
  - The director fires the actors
- Receiver
  - Defines the semantics of the port buffers



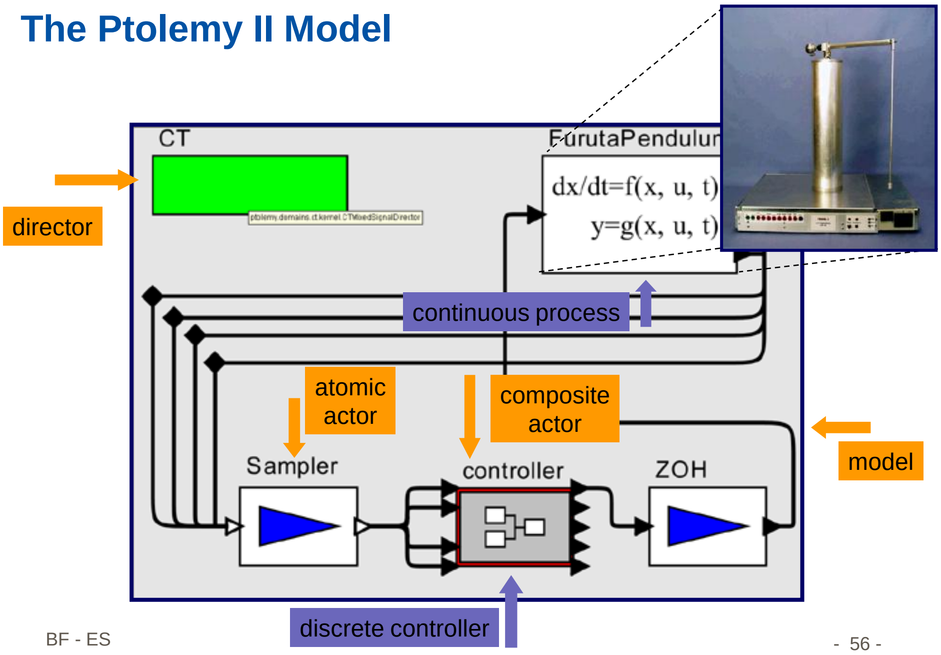


Example: Inverted Pendulum

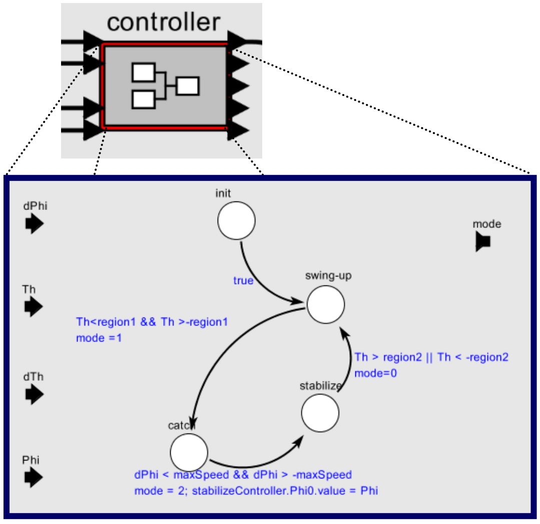
- Classic control problem
- Swing up the pendulum and then keep it in the upright position

Heterogeneous Modeling and Design of Control Systems, Liu/Liu/Eker/Lee, 2003

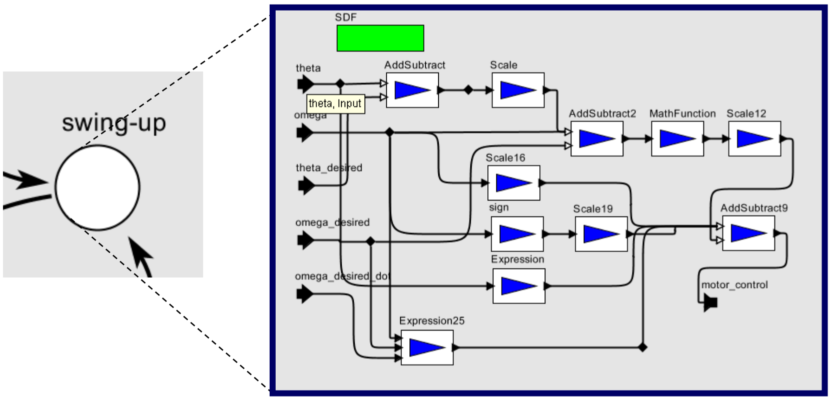
The Ptolemy II Model



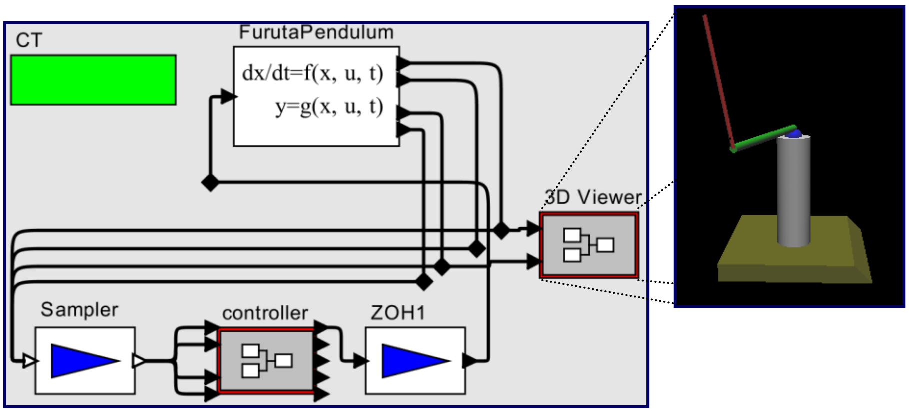
Controller Logic in FSM - Finite State Machine


controller
dPhi
Th
dTh
Phi
mode
init
swing-up
stabilize
catch
true
Th<region1 && Th >-region1
mode =1
Th > region2 || Th < -region2
mode=0
dPhi < maxSpeed && dPhi > -maxSpeed
mode = 2; stabilizeController.Phi0.value = Phi

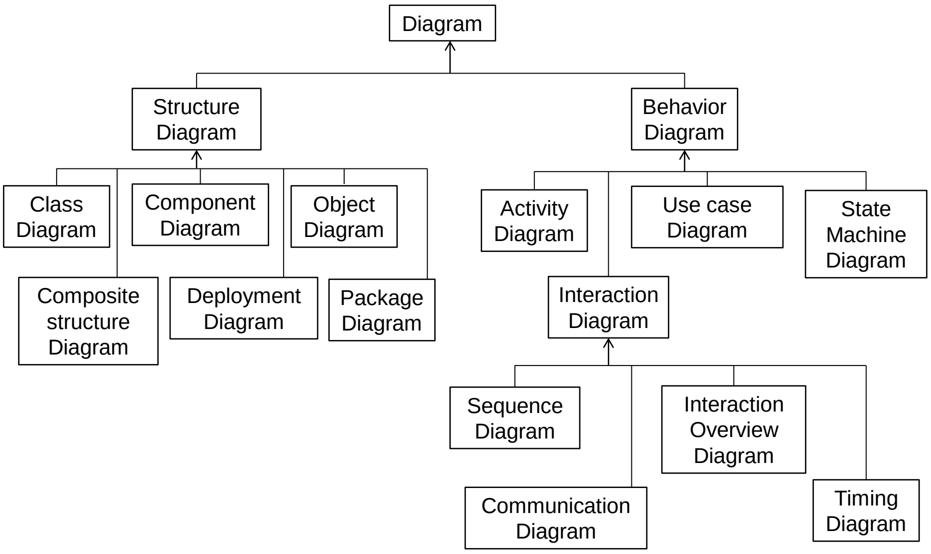
Subcontrollers in SDF - Synchronous Data flow



Visualization in GR - Graphics Domain



UML 2.0 diagram hierarchy



Diagram

Structure Diagram

Behavior Diagram

Class Diagram

Component Diagram

Object Diagram

Composite structure Diagram

Deployment Diagram

Package Diagram

Activity Diagram

Use case Diagram

State Machine Diagram

Interaction Diagram

Sequence Diagram

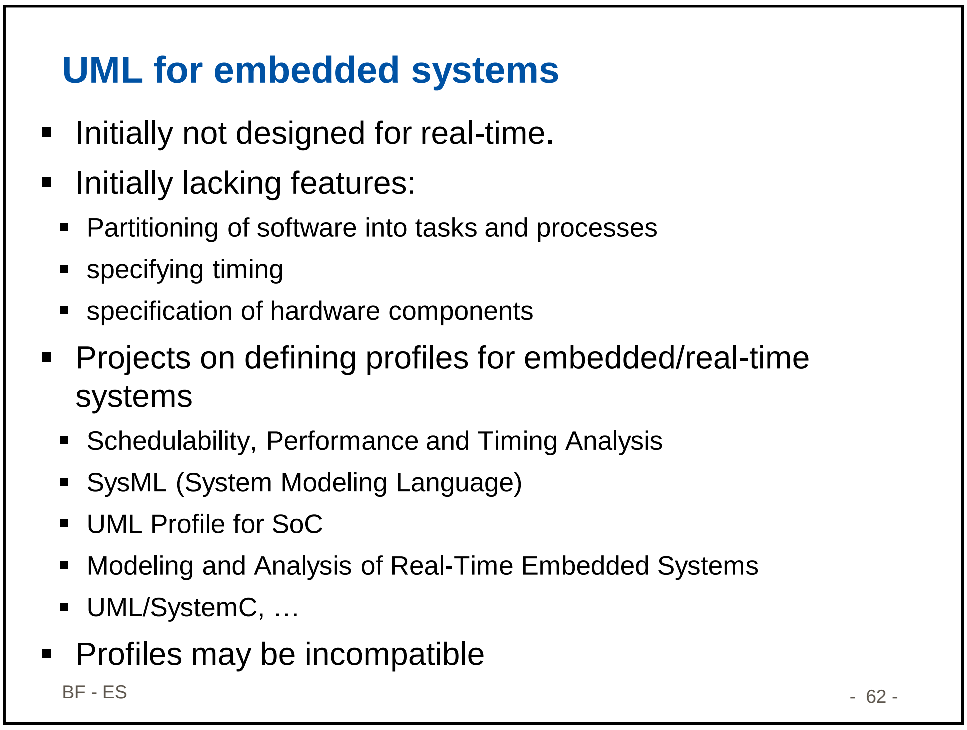
Interaction Overview Diagram

Communication Diagram

Timing Diagram

UML
(Focus on support of early design phases)

Communication/ local computations	**Shared memory**	**Asynchronous message passing**
Communicating finite state machines	State diagrams	
Data flow		Activity diagrams, sequence diagrams, timing diagrams



UML for embedded systems

- Initially not designed for real-time.
- Initially lacking features:
  - Partitioning of software into tasks and processes
  - specifying timing
  - specification of hardware components
- Projects on defining profiles for embedded/real-time systems
  - Schedulability, Performance and Timing Analysis
  - SysML (System Modeling Language)
  - UML Profile for SoC
  - Modeling and Analysis of Real-Time Embedded Systems
  - UML/SystemC, …
- Profiles may be incompatible