Using Synchronous Models for the Design of Parallel Embedded Systems

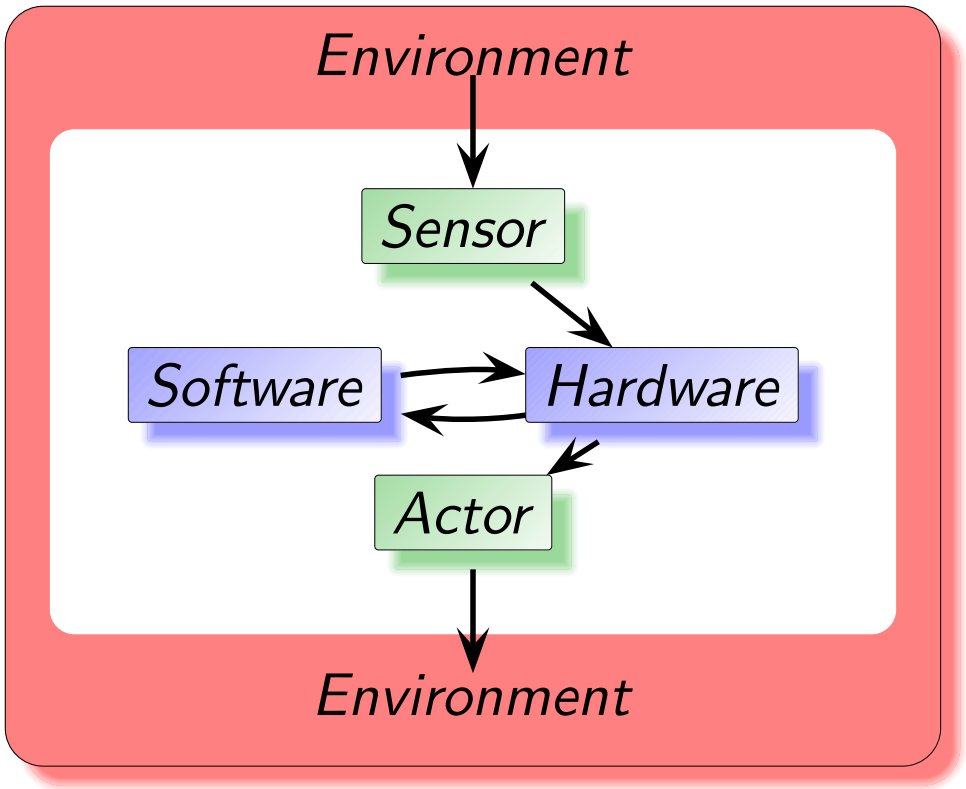
Prof. Dr. Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern

Erlangen, May 24th, 2013

Outline

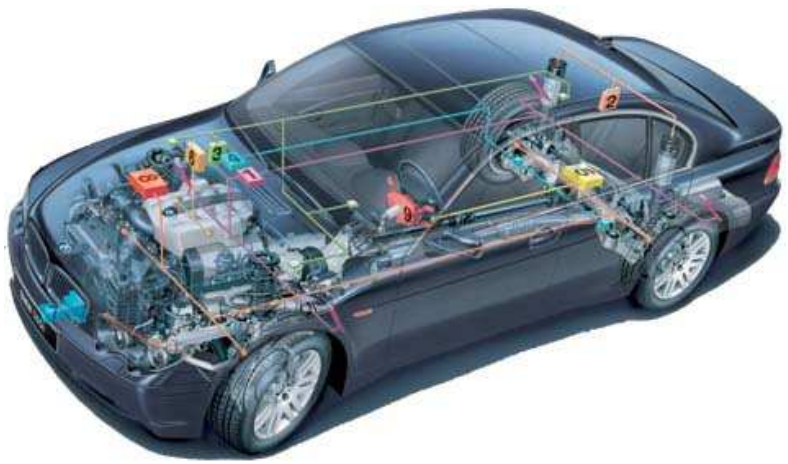
Definition: Embedded Systems



- *direct and ongoing* interaction with environment
- reactive system: if interactions are invoked by environment

$\Rightarrow$ interactions as basic computation steps

Example: Automotive Embedded Systems (ES)



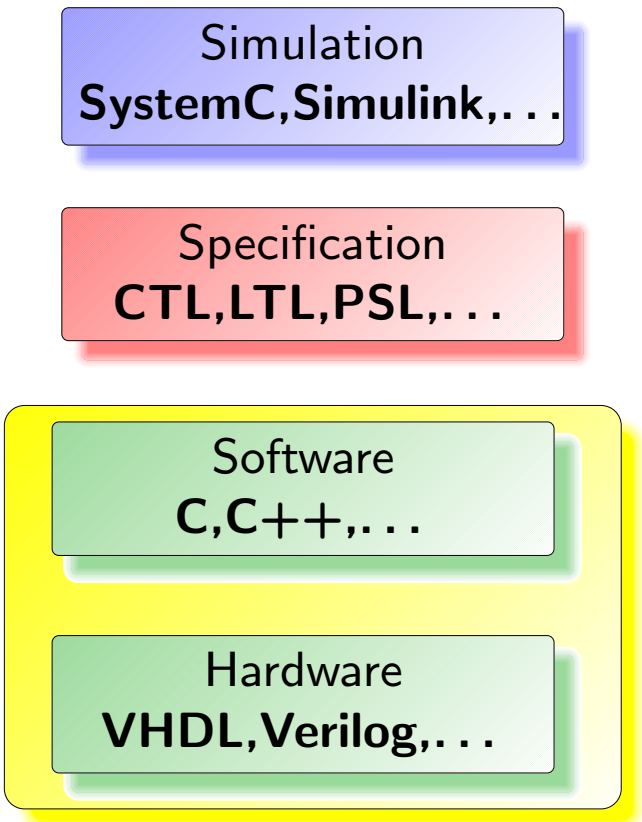
- up to 100 ES in modern cars
- code size grows with a factor of 10 every four years
- 90% of innovations in cars by ES
- $\approx$ 30% development costs due to ES
- 98% of microprocessors in ES

$\rightsquigarrow$ **enormous and still growing economic importance**

Design Problems

- **functional correctness** $\rightsquigarrow$ formal verification
- moreover: **non-functional properties**
  - energy consumption, weight, size
  - real-time capabilities
  - reliability and fault tolerance
- and **heterogeneous computer architectures**
  - multiprocessors with weak memory models
  - application-specific instruction sets
  - HW/SW integration
  - digital/analog components

Many Languages – The Tower of Babel



problems:

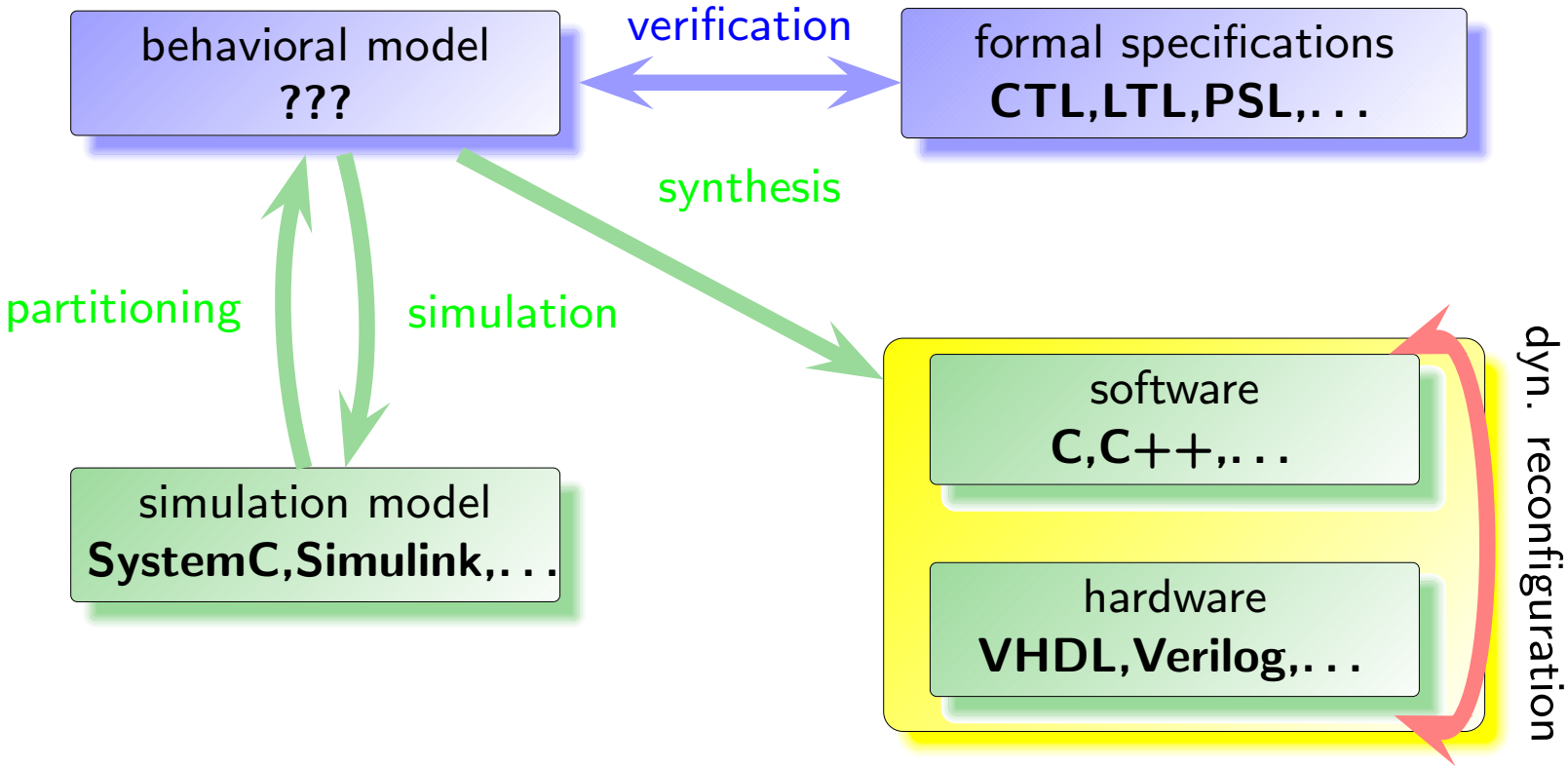
- many languages and architectures
- no unique design methodology
- manual re-implementations

⇝ potential source of errors
⇝ high development costs
⇝ bad re-use of components

solution: model-based design

- unique system model
- automatic translations

Goal: Model-based Design



Idea of Model-based Design

- use system model independent of later architecture
- translate it for particular purposes like
  - modeling: concurrent components with notion of time
  - simulation: deterministic, efficient, ...
  - verification: formal semantics, ...
  - analysis: formal semantics with time/resources,...
  - synthesis: automatic HW- and SW synthesis, ...

$\rightsquigarrow$ design space exploration for optimization

$\rightsquigarrow$ need of a clear semantics/simple analyses

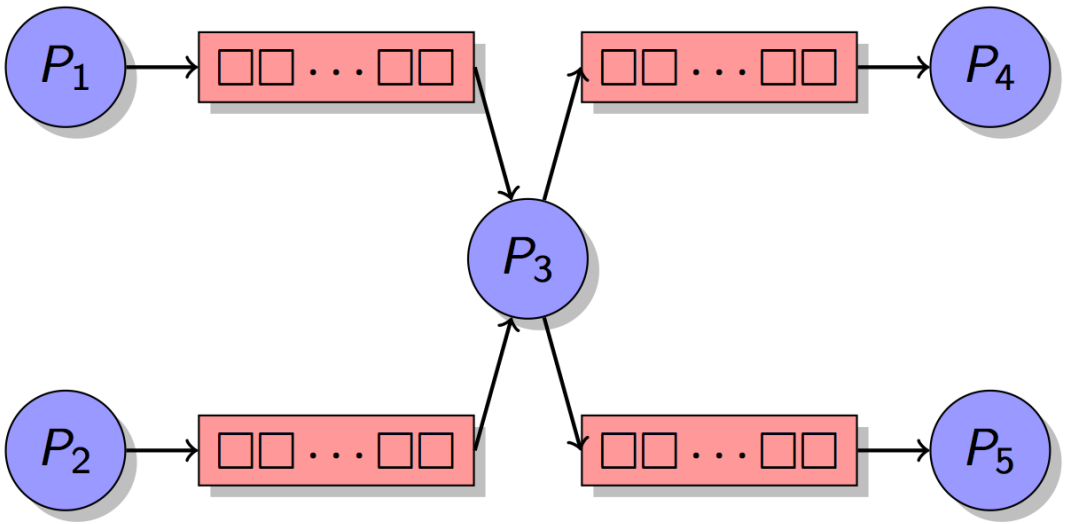
- **models of computation (MoC) [7, 3, 4]**
  explains: **why, when, which atomic actions are executed**

Main Models of Computations

why, when, which atomic actions are executed:

1. data-driven systems: e.g. dataflow process networks
2. event-driven systems: e.g. hardware description languages
3. clock-driven systems: e.g. synchronous languages

Dataflow Process Networks (DPNs)



- sequential processes P_i communicate via FIFO buffers
- FIFOs avoid synchronization of processes
  i.e. reading can be done later than writing the data

Operational Behavior

- e.g. given by firing rules of the process nodes
- nodes *can* fire, but do not have to fire

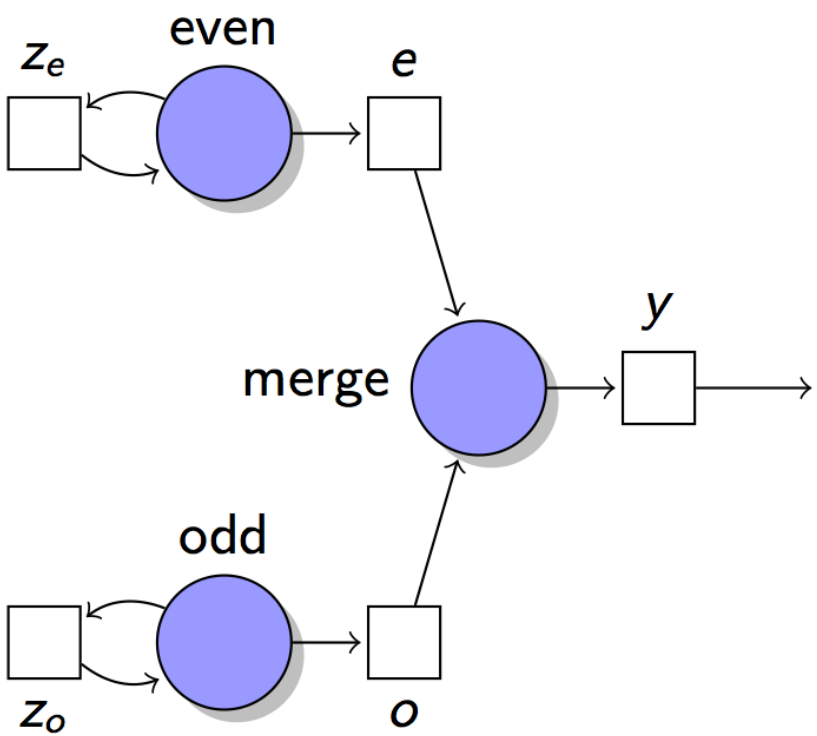
$\rightsquigarrow$ no deterministic schedule for firing the nodes

- but: same input streams should produce same output streams

$\rightsquigarrow$ **stream processing functions**

- however, this determinism is not always given (next slide)

Example DPN



firing rules of merge

x_1	x_2	y
$(a :: A)$	$(b :: B)$	$[a, b]$
$[]$	$(b :: B)$	$[b]$
$(a :: A)$	$[]$	$[a]$

DPN as equations

$$\begin{cases} (e, z_e) = \text{even}(z_e) \\ (o, z_o) = \text{odd}(z_o) \\ y = \text{merge}(e, o) \end{cases}$$

Problem: Nondeterminism

- behavior 1: all nodes fire asap

$$\begin{pmatrix} e \mapsto [] \\ o \mapsto [] \\ y \mapsto [] \\ z_e \mapsto [] \\ z_o \mapsto [] \end{pmatrix} \overset{\text{odd, even}}{\rightsquigarrow} \begin{pmatrix} e \mapsto [0] \\ o \mapsto [1] \\ y \mapsto [] \\ z_e \mapsto [0] \\ z_o \mapsto [1] \end{pmatrix} \overset{\text{odd, even, merge}}{\rightsquigarrow} \begin{pmatrix} e \mapsto [2] \\ o \mapsto [3] \\ y \mapsto [0,1] \\ z_e \mapsto [2] \\ z_o \mapsto [3] \end{pmatrix} \overset{\text{odd, even, merge}}{\rightsquigarrow} \begin{pmatrix} e \mapsto [4] \\ o \mapsto [5] \\ y \mapsto [0,1,2,3] \\ z_e \mapsto [4] \\ z_o \mapsto [5] \end{pmatrix} \ldots$$

- behavior 2: node 'even' does not fire at all

$$\begin{pmatrix} e \mapsto [] \\ o \mapsto [] \\ y \mapsto [] \\ z_e \mapsto [] \\ z_o \mapsto [] \end{pmatrix} \overset{\text{odd}}{\rightsquigarrow} \begin{pmatrix} e \mapsto [] \\ o \mapsto [1] \\ y \mapsto [] \\ z_e \mapsto [] \\ z_o \mapsto [1] \end{pmatrix} \overset{\text{odd, merge}}{\rightsquigarrow} \begin{pmatrix} e \mapsto [] \\ o \mapsto [3] \\ y \mapsto [1] \\ z_e \mapsto [] \\ z_o \mapsto [3] \end{pmatrix} \overset{\text{odd, merge}}{\rightsquigarrow} \begin{pmatrix} e \mapsto [] \\ o \mapsto [5] \\ y \mapsto [1,3] \\ z_e \mapsto [] \\ z_o \mapsto [5] \end{pmatrix} \ldots$$

Enforcing Determinism (Kahn [5])

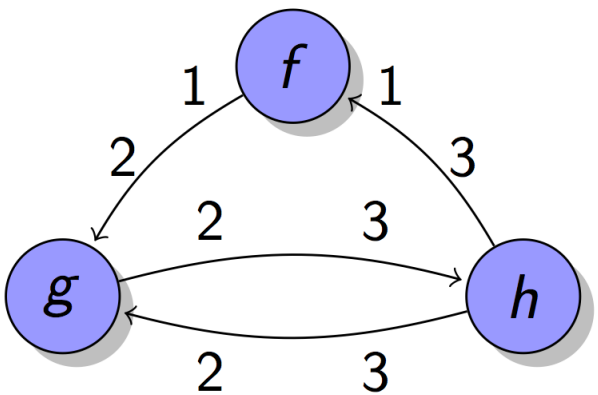
- Kahn's DPNs [5]
  - K1: **no emptiness checks**:
    number of values in buffers must not be checked for firing
  - K2: **use blocking read**:
    reading a value from an empty buffer must wait for values
  - K3: **use sequential functions**: will be considered later
- infinite computations moreover demand fairness:
  each node that can fire, must eventually do so

Boundedness of DPNs

- **boundedness** = finite memory for FIFO buffers
  - boundedness is undecidable in general
  - but decidable for special DPNs
- **static DPNs**
  - always consume the same number of values from an input x
  - and produce the same number of values for an output y
  - may be different for other inputs x' or other outputs y'
- **cyclo-static DPNs**
  - consumption/production numbers change periodically

Example: Boundedness of DPNs

- **problem: determine static schedule for infinite repetition**
- let r_f, r_g, r_h be the number of firings of nodes f, g, h
- edges in DPN on the left are number of produced and consumed values in FIFO on the edge



$\rightsquigarrow$ balance equations

$$\begin{pmatrix} 1 & -2 & 0 \\ 0 & -2 & 3 \\ 0 & 2 & -3 \\ -1 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} r_f \\ r_g \\ r_h \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- solution $(r_f, r_g, r_h) = (6, 3, 2) \cdot \lambda$
- it remains to schedule these firings

Main Models of Computations

why, when, which atomic actions are executed:

1. data-driven systems: e.g. dataflow process networks
2. event-driven systems: e.g. hardware description languages
3. clock-driven systems: e.g. synchronous languages

Discrete Event Systems

- originally developed for efficient simulation
- **system = set of sequential processes $P_1,\ldots,P_m$**
  - communication over shared variables
  - processes have statements to wait on events, i.e.:
    - a condition becomes true
    - a point of time has been reached
    - the value of a variable has been changed
  - process will be activated if its wait condition becomes true
  - then: its code is 'elaborated' up to the next wait condition
  - i.e., assignments $x = \tau$ are noted in a schedule $\mathcal{S}$

Simulation Semantics

- **discrete event MoC is defined by a simulator**
  - determine next event based on schedule $\mathcal{S}$
  - determine activated processes and elaborate these
  - $\rightsquigarrow$ repeat with new schedule $\mathcal{S}'$

$\rightsquigarrow$ **computation is driven by occurrence of events**

- example languages:
  VHDL, Verilog, SystemC, SystemVerilog, Simulink, ...

Example: VHDL

- process of a VHDL program:

P_1 : process
  $x \Leftarrow$ transport $x + 2$ after 0 ns;
  $x \Leftarrow$ transport $x + 3$ after 2 ns;
  wait on x;
end process

- simulation step 1:
  $\mathcal{E} = \{(x, 0)\}$, $\mathcal{S} := \{\}$, $t_{\text{curr}} = 0\text{ns}$
  $\Rightarrow$ schedule $\mathcal{S} := \{(0\text{ns}, x, 2), (2\text{ns}, x, 3)\}$
- simulation step 2:
  $\mathcal{E} = \{(x, 2)\}$, $\mathcal{S} := \{(2\text{ns}, x, 3)\}$, $t_{\text{curr}} = 0\text{ns}$
  $\Rightarrow$ schedule $\mathcal{S} := \{(0\text{ns}, x, 4), \cancel{(2\text{ns}, x, 3)}, (2\text{ns}, x, 5)\}$

Example: VHDL

- process of a VHDL program:

P_1 : process
  $x \Leftarrow$ transport $x + 2$ after 0 ns;
  $x \Leftarrow$ transport $x + 3$ after 2 ns;
  wait on x;
end process

- simulation step 3:
  $\mathcal{E} = \{(x, 4)\}$, $\mathcal{S} := \{(2\text{ns}, x, 5)\}$, $t_{\text{curr}} = 0\text{ns}$
  $\Rightarrow$ schedule $\mathcal{S} := \{(0\text{ns}, x, 6), \sout{(2\text{ns}, x, 5)}, (2\text{ns}, x, 7)\}$
- simulation step i:
  $\mathcal{E} = \{(x, 2i)\}$, $\mathcal{S} := \{(2\text{ns}, x, 2i + 1)\}$, $t_{\text{curr}} = 0\text{ns}$
  $\Rightarrow \mathcal{S} := \{(0\text{ns}, x, 2i + 2), \sout{(2\text{ns}, x, 2i + 1)}, (2\text{ns}, x, 2i + 3)\}$
- note: insertion of $(0\text{ns}, x, 2i + 2)$ removes $(2\text{ns}, x, 2i + 1)$
- otherwise: schedule would grow unboundedly

Semantic Problems

- **two-dimensional time**
  - several simulation steps may refer to the same physical point of time
  - variables may have several values at one point of time
- **semantic problems**
  - schedule $\mathcal{S}$ may be unbounded
  - physical time may stop while simulation proceeds
  - processes may suffer from deadlocks and livelocks
- **solutions may be analogous to synchronous systems**

Main Models of Computations

why, when, which atomic actions are executed:

1. data-driven systems: e.g. dataflow process networks
2. event-driven systems: e.g. hardware description languages
3. clock-driven systems: e.g. synchronous languages

Synchronous Systems

- modules have
  - inputs $x_1, \ldots, x_m$
  - outputs $y_1, \ldots, y_n$
  - and internal state variables $z_1, \ldots, z_k$
- **computation by discrete (reaction) steps:**
  - reactions are driven by clock ticks
  - read all inputs
  - compute output values and next internal state
- **distinction between micro and macro steps**
  - macro step = reaction = variable assignment
  - macro steps consist of finitely many micro steps

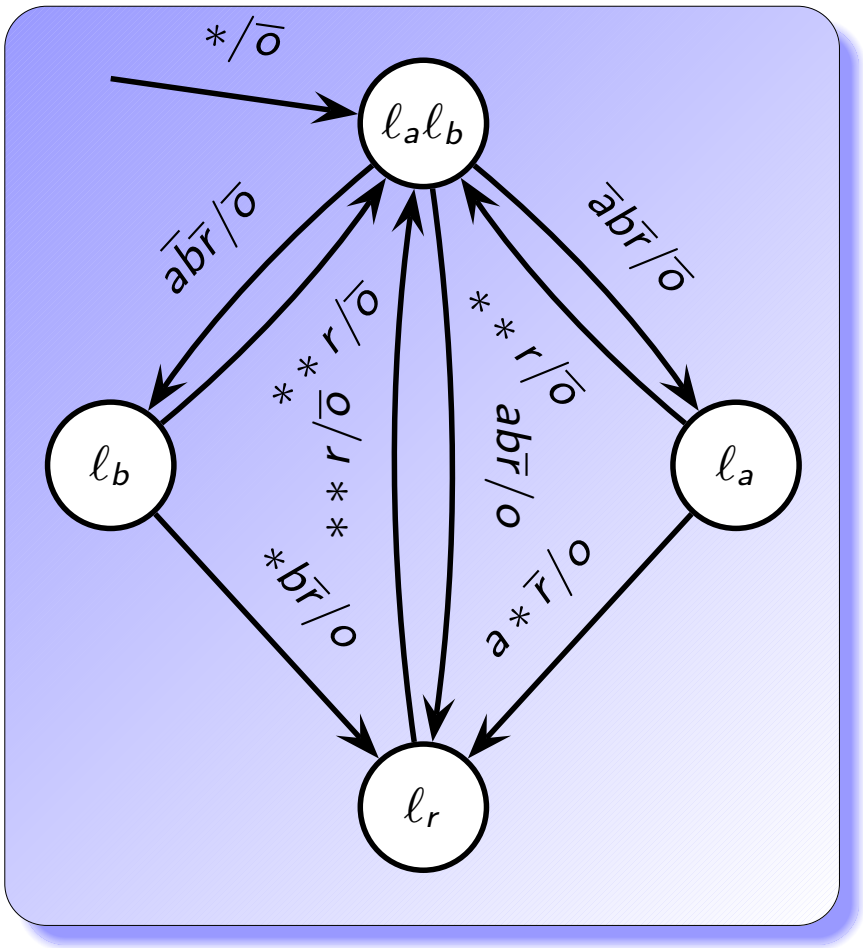
  $\rightsquigarrow$ all micro steps are executed on the same variable assignment

Example: Quartz Language

`nothing`	(empty statement)
ℓ : `pause`	(macro step)
$x = \tau$, `next`$(x) = \tau$	(assignments)
`if`(σ) S_1 `else` S_2	(conditional)
S_1 `;` S_2	(sequence)
$S_1 \parallel S_2$	(concurrency)
`do` S `while`(σ)	(loop)
`[weak] [immediate] abort S when`(σ)	(abortion)
`[weak] [immediate] suspend S when`(σ)	(suspension)
$\{\alpha\ x;\ S\}$	(local variable)

Example: Quartz Program

```
module ABRO(?a,?b,?r,!o) {
loop
  abort {
    {
      ℓa: await(a);
      ||
      ℓb: await(b);
    }
    o = true;
    ℓr: await(r);
  } when(r)
}
```



Causal Execution of Micro Steps

- example: synchronous program

$$\left[\begin{array}{l} \texttt{b = true;} \\ p : \texttt{pause;} \\ \texttt{if(a) b = true;} \\ r : \texttt{pause} \end{array}\right] \Big\| \left[\begin{array}{l} q : \texttt{pause;} \\ \texttt{if(!b) c = true;} \\ \texttt{a = true;} \\ s : \texttt{pause} \end{array}\right]$$

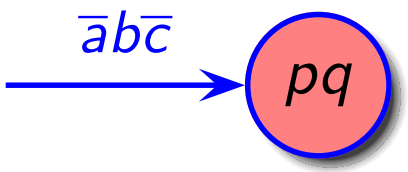
- equivalent automaton:

Causal Execution of Micro Steps

- example: synchronous program

$$\left[\begin{array}{l} \texttt{b = true;} \\ p\,\texttt{: pause;} \\ \texttt{if(a) b = true;} \\ r\,\texttt{: pause} \end{array}\right] \Big\| \left[\begin{array}{l} q\,\texttt{: pause;} \\ \texttt{if(!b) c = true;} \\ \texttt{a = true;} \\ s\,\texttt{: pause} \end{array}\right]$$

- equivalent automaton:

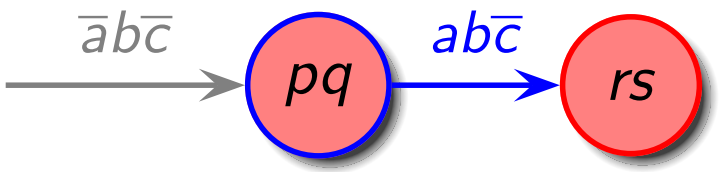


Causal Execution of Micro Steps

- example: synchronous program

```
[ b = true;          ]|||[ q : pause;          ]
[ p : pause;         ]   [ if(!b) c = true;    ]
[ if(a) b = true;    ]   [ a = true;           ]
[ r : pause          ]   [ s : pause           ]
```

- equivalent automaton:

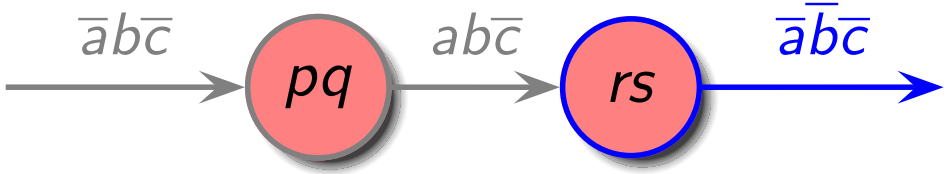


Causal Execution of Micro Steps

- example: synchronous program

$$\left[\begin{array}{l} \texttt{b = true;} \\ p\,\texttt{: pause;} \\ \texttt{if(a) b = true;} \\ r\,\texttt{: pause} \end{array}\right] \Big\| \left[\begin{array}{l} q\,\texttt{: pause;} \\ \texttt{if(!b) c = true;} \\ \texttt{a = true;} \\ s\,\texttt{: pause} \end{array}\right]$$

- equivalent automaton:

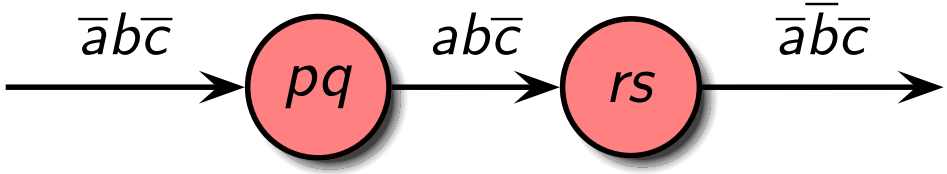


Causal Execution of Micro Steps

- example: synchronous program

$$\left[\begin{array}{l} \texttt{b = true;} \\ p : \texttt{pause;} \\ \texttt{if(a) b = true;} \\ r : \texttt{pause} \end{array}\right] \Big\| \left[\begin{array}{l} q : \texttt{pause;} \\ \texttt{if(!b) c = true;} \\ \texttt{a = true;} \\ s : \texttt{pause} \end{array}\right]$$

- equivalent automaton:



Formal Semantics

- **causal execution:** do not read variables that will be written, but have not already been written in the macro step
- is formally defined for Quartz by SOS rules (Plotkin 1981)

$\rightsquigarrow$ directly defines a simulator

- **main idea**
  - introduce value $\perp$: means 'not yet known'
  - initially: all inputs known, all outputs unknown
  - estimate:
    - $\mathcal{D}_{\text{must}}$: set of all actions that must be fired
    - $\mathcal{D}_{\text{can}}$: set of all actions that can be fired
  - and refine known values

Consistency Checks of MoCs

- **data-driven MoCs [5, 6]:**
  - check determinism (e.g. Kahn's rules)
  - check boundedness of buffers
- **event-driven MoCs [2]:**
  - check boundedness of schedule
  - check absence of deadlocks and livelocks
- **clock-driven MoCs [1]:**
  - check causality = check sequential execution
  - check clock consistency if several clocks are used

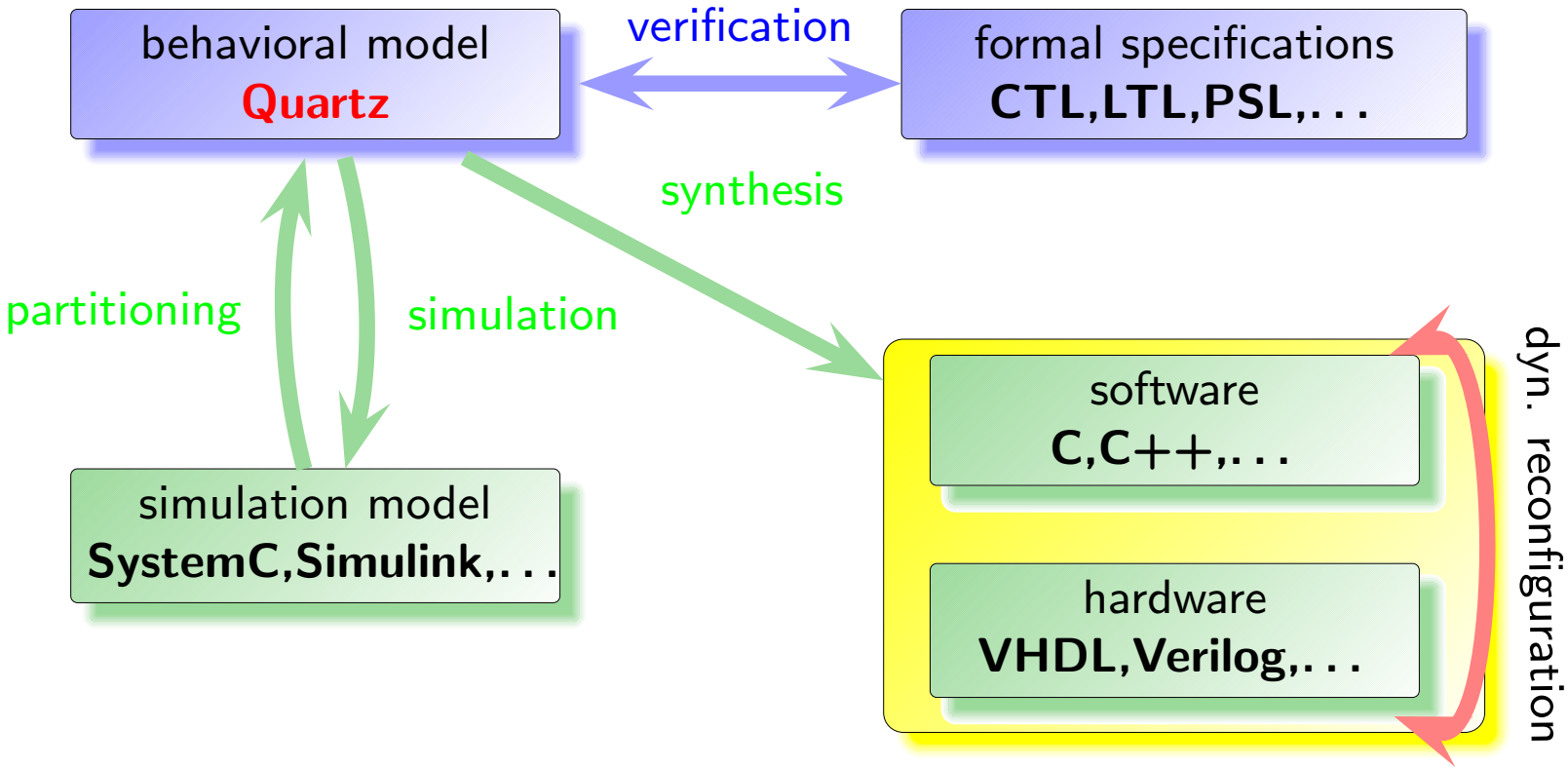
$\rightsquigarrow$ **then, deterministic finite-state systems are obtained**

Advantages of MoCs

- **particular uses of MoCs**
  - simulation: event-driven MoCs
  - verification: clock-driven MoCs
  - HW-Synthesis: clock-driven MoCs
  - SW-Synthesis (multithreaded): communicating threads
  - distributed systems: data-driven MoCs
- **transformations between MoCs are required!!**

Model-based Design Using Averest

- Averest is a model-based design tool
- developed at the U. Kaiserslautern (www.averest.org)



Intermediate Representation by Guarded Actions

- intermediate representation of MoCs required
- Lee and Sangiovanni-Vincentelli [7]: tagged tokens
- **in Averest: guarded actions** (γ, α)
  - trigger condition γ with atomic action α
  - (γ, α) is enabled, if γ holds
- **guarded actions reduce languages to their MoC**
  - recall MoC: when, why, which action is executed
  - γ is the reason (why?) for executing α (which?)
  - 'when' is defined as:
    - synchronous MoC: execute **all enabled** actions
    - asynchronous MoC: execute **some enabled** actions

Translation to Guarded Actions

- idea: determine condition γ for each action α
- however: very difficult to do
  - distinction between surface and depth required
  - modular translation very difficult due to reincarnations etc.

$\rightsquigarrow$ translation has been formally verified

- programs of size n may yield $O(n^2)$ many guarded actions

Causality Analysis on Guarded Actions

- causality analysis can be directly done on guarded actions
- using Brzozowski-Seger's ternary simulation:

$\wedge$	$\bot$	0	1
$\bot$	$\bot$	0	$\bot$
0	0	0	0
1	$\bot$	0	1

$\vee$	$\bot$	0	1
$\bot$	$\bot$	$\bot$	1
0	$\bot$	0	1
1	1	1	1

x	$\neg x$
$\bot$	$\bot$
0	1
1	0

- all functions are monotonic w.r.t. $\bot \sqsubseteq 1, 0$
- causality analysis done by computing least fixpoint

Example

```
module P15(!o1,!o2) {
 o2 = true;
 if(o1)
  if(!o2)
   o1 = true;
}
```

- guarded actions

$$\begin{array}{lcl} o_1 \wedge \neg o_2 & \Rightarrow & o_1 \\ 1 & \Rightarrow & o_2 \end{array}$$

- causality analysis:

	0	1	2
o_1	$\bot$	$\bot$	0
o_2	$\bot$	1	1

$\rightsquigarrow$ program is causal

Equivalent Problems

- stability of asynchronous circuits (ternary simulation)
  - check whether all signals stabilize for all input values
  - independent of delay time of the gates
- deadlock freedom of parallel programs
  - threads wait on each other
  - problem: check whether deadlock can occur
- proofs in intuitionistic logic
  - tertium non datur $(x \vee \neg x)$ cannot be proved
  - all proof must be constructive
- three-valued logic
  - models progress of micro step execution
  - $\perp$: means not yet known

Hardware Synthesis

- consider guarded actions of $\mathtt{x}$
  - $(\chi_1, \mathtt{next(x)} = \pi_1), \ldots, (\chi_q, \mathtt{next(x)} = \pi_q)$
  - $(\gamma_1, \mathtt{x} = \tau_1), \ldots, (\gamma_p, \mathtt{x} = \tau_p)$

$\rightsquigarrow$ compute equations with carrier variable $\mathtt{x'}$:

$$\mathtt{x} = \left(\begin{array}{l} \mathtt{case} \\ \quad \gamma_1 : \tau_1; \\ \quad \vdots \\ \quad \gamma_p : \tau_p; \\ \mathtt{else\ x'} \end{array} \right) \quad \mathtt{next(x')} = \left(\begin{array}{l} \mathtt{case} \\ \quad \chi_1 : \pi_1; \\ \quad \vdots \\ \quad \chi_q : \pi_q; \\ \mathtt{else\ Default(x)} \end{array} \right)$$

$\rightsquigarrow$ $\mathtt{x'}$ stores delayed assignments of previous step

Synthesis of Sequential Software

- **hardware synthesis via equation systems**
  - one equation for each output and state variable
  - requires $O(n^2)$ gates
  - in each cycle, the complete equation system must be evaluated
  - optimization by high-level synthesis
- **sequential software**
  - evaluation of all equations per macro step
  - causal order must be respected
  - alternative: compute EFSM and precompute equations per control state

  $\rightsquigarrow$ potential exponential growth of code, but much faster

Synthesis of Parallel Software

- **multithreaded software is asynchronous!**

⇝ **translate guarded actions to DPN**

  - one node P_x for each output/state variable x
  - node P_x computes the value of x in each macro step

⇝ **static DPN:**

  - each node will fire once per macro step
  - causality ensures existence of schedule
  - clock is generated if all values are available

- **in practice: often too slow**
  - synchronization of nodes enforced by generated clock
  - nodes typically wait a long time for new values
  - better: translation to asynchronous systems

Optimization 1: Elimination of Passive Code

- **observation**
  - not all values are not needed in some macro steps
  - example: if $x = 0$ holds, then y is not required for $z = x \wedge y$

$\rightsquigarrow$ **introduce new value $\boxdot$ for analysis**

  - $\boxdot$ is a placeholder for a concrete, but unwanted value
  - $\boxdot$ will not be computed and also not communicated

- **compiler optimization:**
  - compute for every (γ, α) a condition β, such that $(\gamma \wedge \beta, \alpha)$ does not change behavior
  - analogous to classic dataflow analysis in compilers

Optimization 2: Replace Clock- by Data-Triggers

- **synchronous systems are driven by clocks**
- the clock is not needed if all values have to be generated in every cycle (then, nodes are simply driven by data)
- however, if passive values are suppressed, then a clock would be again required
- **better: endochronous systems**
  - these are synchronous systems that can generate their own local clock
  - value of one input implicitly encodes which other values are required

Example: if-then-else node

x_1	x_2	x_3	y
$(1 :: A)$	$(b :: B)$	$(c :: C)$	$[b]$
$(0 :: A)$	$(b :: B)$	$(c :: C)$	$[c]$

x_1	x_2	x_3	y
$(1 :: A)$	$(b :: B)$	C	$[b]$
$(0 :: A)$	B	$(c :: C)$	$[c]$

$\xi(x_1)$:	1	0	0	1	1	...
$\xi(x_2)$:	1	3	5	7	9	...
$\xi(x_3)$:	0	2	4	6	8	...
$\xi(y)$:	1	2	4	7	9	...

$\xi(x_1)$:	1	0	0	1	1	...
$\xi(x_2)$:	1	$\boxdot$	$\boxdot$	7	9	...
$\xi(x_3)$:	$\boxdot$	2	4	$\boxdot$	$\boxdot$	...
$\xi(y)$:	1	2	4	7	9	...

$\rightsquigarrow$ "'if-then-else"' is endochronous

- x_1 is always read, and determines whether x_2 or x_3 will be read

Sequential Functions

- sequential functions
  - always read one input x_1
  - based on the value read, decide which input to read next
  - until enough values were read to fire the node
- the following (Gustave) function is not sequential

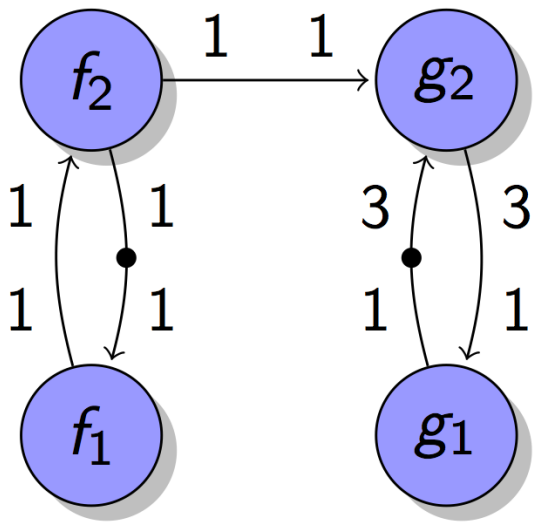
x_1	x_2	x_3	y
$(1 :: A)$	$(0 :: B)$	C	[1]
A	$(1 :: B)$	$(0 :: C)$	[1]
$(0 :: A)$	B	$(1 :: C)$	[1]

$\rightsquigarrow$ **desynchronization**

- generate DPN with sequential functions from synchronous guarded actions
- these DPNs can be run asynchronously

$\rightsquigarrow$ **latency insensitive design/elastic circuits**

Boundedness for Parallel DPNs



- topology matrix yields the following solutions

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & -1 & 3 \end{pmatrix} \quad r = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 1 \end{pmatrix} \lambda$$

- partition into two sub-systems $S_1 := \{f_1, f_2\}$ and $S_2 := \{g_1, g_2\}$

$\rightsquigarrow$ buffer (f_2, g_2) can overflow

$\rightsquigarrow$ 'backpressure' edge from S_2 to S_1 required

$\rightsquigarrow$ not necessary if $\mathcal{T}$ has S- and T-invariants

Model-based Design Using Averest

- **system behavior given by synchronous program**
  - precise formal semantics $\rightsquigarrow$ formal verification possible
  - deterministic/reproducible simulation

  $\rightsquigarrow$ simplified WCET analysis
- **internal representation by guarded actions**
  - reduce model to core of its MoC
  - efficient causality analysis
  - translation to (elastic) synchronous hardware circuits
  - translation to sequential software
  - translation to parallel software (asynchronous DPN)

Design Tools Using Different MoCs

- Ptolemy (Berkeley, USA): http://ptolemy.eecs.berkeley.edu/
- Metropolis (Berkeley, USA): http://embedded.eecs.berkeley.edu/metropolis/
- ForSyDe (KTH, Schweden): http://www.ict.kth.se/forsyde/
- SysteMoC (Erlangen): http://www12.cs.fau.de/research/scd/systemoc.php
- SysML und UML/MARTE
- Averest (Kaiserslautern): http://www.averest.org

References and Further Reading I

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone.
The synchronous languages twelve years later.
Proceedings of the IEEE, 91(1):64–83, 2003.

[2] C.G. Cassandras and S. Lafortune.
Introduction to Discrete Event Systems.
Springer, 2 edition, 2008.

[3] A. Girault, B. Lee, and E.A. Lee.
Hierarchical finite state machines with multiple concurrency models.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD), 18(6):742–760, June 1999.

[4] A. Jantsch.
Modeling Embedded Systems and SoCs.
Morgan Kaufmann, 2004.

References and Further Reading II

[5] G. Kahn.
The semantics of a simple language for parallel programming.
In J.L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, 1974. North-Holland.

[6] E.A. Lee.
Consistency in dataflow graphs.
IEEE Transactions on Parallel and Distributed Systems (TPDS), 2(2), 1991.

[7] E.A. Lee and A. Sangiovanni-Vincentelli.
A framework for comparing models of computation.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD), 17(12):1217–1229, December 1998.

[8] G.D. Plotkin.
LCF considered as a programming language.
Theoretical Computer Science (TCS), 5(3):223–255, December 1977.