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

1 Motivation: Model-based Design

2 Models of Computation
   - Data-Driven MoCs
   - Event-Driven MoCs
   - Clock-Driven MoCs

3 The Averest Tool
   - Translation to Guarded Actions
   - Causality Analysis
   - Hardware and Software Synthesis
   - Synthesis of Parallel Software

4 Summary
**Definition: Embedded Systems**

- **Direct and ongoing interaction** with environment
- **Reactive system**: if interactions are invoked by environment

⇒ 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

\( \rightarrow \) enormous and still growing economic importance
**Design Problems**

- **functional correctness** $\leadsto$ 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

- **behavioral model**
  - ???
- **formal specifications**
  - CTL, LTL, PSL, ...
- **simulation model**
  - SystemC, Simulink, ...
- **software**
  - C, C++, ...
- **hardware**
  - VHDL, Verilog, ...

**Processes:**
- Verification
- Synthesis
- Partitioning
- Simulation
- Dynamic reconfiguration
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, . . .

→ design space exploration for optimization
→ 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
  - no deterministic schedule for firing the nodes
  - but: same input streams should produce same output streams
  - stream processing functions
  - however, this determinism is not always given (next slide)
Example DPN

**firing rules of merge**

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a :: A)$</td>
<td>$(b :: B)$</td>
<td>$[a, b]$</td>
</tr>
<tr>
<td>$[]$</td>
<td>$(b :: B)$</td>
<td>$[b]$</td>
</tr>
<tr>
<td>$(a :: A)$</td>
<td>$[]$</td>
<td>$[a]$</td>
</tr>
</tbody>
</table>

**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

  \[
  \begin{align*}
  e &\mapsto [0] \\
  o &\mapsto [1] \\
  y &\mapsto [] \\
  z_e &\mapsto [0] \\
  z_o &\mapsto [1]
  \end{align*}
  \]

- behavior 2: node ‘even’ does not fire at all

  \[
  \begin{align*}
  e &\mapsto [] \\
  o &\mapsto [1] \\
  y &\mapsto [] \\
  z_e &\mapsto [] \\
  z_o &\mapsto [1]
  \end{align*}
  \]
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

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

- Solution \((r_f, r_g, r_h) = (6, 3, 2) \cdot \lambda\)
- It remains to schedule these firings
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 $S$
Simulation Semantics

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

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

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

- process of a VHDL program:

  \[ P_1 : \text{process} \]
  \[ x \leftarrow \text{transport } x + 2 \text{ after } 0 \text{ ns}; \]
  \[ x \leftarrow \text{transport } x + 3 \text{ after } 2 \text{ ns}; \]
  \[ \text{wait on } x; \]
  \[ \text{end process} \]

- simulation step 1:
  \[ E = \{(x, 0)\}, S := \{\}, t_{\text{curr}} = 0\text{ns} \]
  \[ \Rightarrow \text{schedule } S := \{(0\text{ns}, x, 2), (2\text{ns}, x, 3)\} \]

- simulation step 2:
  \[ E = \{(x, 2)\}, S := \{(2\text{ns}, x, 3)\}, t_{\text{curr}} = 0\text{ns} \]
  \[ \Rightarrow \text{schedule } S := \{(0\text{ns}, x, 4), (2\text{ns}, x, 3), (2\text{ns}, x, 5)\} \]
Example: VHDL

process of a VHDL program:

\[ P_1 : \text{process} \]
\[ x \leftarrow \text{transport } x + 2 \text{ after } 0 \text{ ns}; \]
\[ x \leftarrow \text{transport } x + 3 \text{ after } 2 \text{ ns}; \]
\[ \text{wait on } x; \]
\[ \text{end process} \]

simulation step 3:
\[ \mathcal{E} = \{(x, 4)\}, \quad \mathcal{S} := \{(2\text{ns}, x, 5)\}, \quad t_{\text{curr}} = 0\text{ns} \]
\[ \Rightarrow \text{schedule } \mathcal{S} := \{(0\text{ns}, x, 6), (2\text{ns}, x, 5), (2\text{ns}, x, 7)\} \]

simulation step \( i \):
\[ \mathcal{E} = \{(x, 2i)\}, \quad \mathcal{S} := \{(2\text{ns}, x, 2i + 1)\}, \quad t_{\text{curr}} = 0\text{ns} \]
\[ \Rightarrow \mathcal{S} := \{(0\text{ns}, x, 2i + 2), (2\text{ns}, x, 2i + 1), (2\text{ns}, x, 2i + 3)\} \]

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 $S$ may be unbounded
  - physical time may stop while simulation proceeds
  - processes may suffer from deadlocks and livelocks

- **solutions may be analogous to synchronous systems**
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
  - all micro steps are executed on the same variable assignment
Example: Quartz Language

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

module ABRO(?a,?b,?r,!o) {
  loop
    abort {
      { 
        $l_a$: await(a); 
        $\parallel$
        $l_b$: await(b);
      } 
      o = true; 
      $l_r$: await(r);
    }
    when(r)
}
Causal Execution of Micro Steps

- example: synchronous program

\[
\begin{align*}
  & b = \text{true}; \\
  & p : \text{pause}; \\
  & \text{if}(a) \ b = \text{true}; \\
  & r : \text{pause}
\end{align*}
\]

\[
\begin{align*}
  & q : \text{pause}; \\
  & \text{if}(!b) \ c = \text{true}; \\
  & a = \text{true}; \\
  & s : \text{pause}
\end{align*}
\]

- equivalent automaton:
Causal Execution of Micro Steps

- example: synchronous program

\[
\begin{align*}
\text{b} &= \text{true}; \\
\text{p} &: \text{pause}; \\
\text{if(a) b} &= \text{true}; \\
\text{r} &: \text{pause}
\end{align*}
\]

\[
\begin{align*}
\text{q} &: \text{pause}; \\
\text{if(!b) c} &= \text{true}; \\
\text{a} &= \text{true}; \\
\text{s} &: \text{pause}
\end{align*}
\]

- equivalent automaton:
Causal Execution of Micro Steps

- example: synchronous program

\[
\begin{align*}
  b &= \text{true; } p : \text{pause; } \text{if}(a) \ b &= \text{true; } r : \text{pause} \\
  q &= \text{pause; } \text{if}(!b) \ c &= \text{true; } a = \text{true; } s : \text{pause}
\end{align*}
\]

- equivalent automaton:
example: synchronous program

\[
\begin{align*}
  b &= \text{true}; \\
  p &: \text{pause}; \\
  \text{if}(a) \ b &= \text{true}; \\
  r &: \text{pause}
\end{align*}
\]

\[
\begin{align*}
  q &: \text{pause}; \\
  \text{if}(\neg b) \ c &= \text{true}; \\
  a &= \text{true}; \\
  s &: \text{pause}
\end{align*}
\]

equivalent automaton:
Causal Execution of Micro Steps

- example: synchronous program

\[
\begin{align*}
&b = \text{true;} \\
&p: \text{pause;} \\
&\text{if(a) } b = \text{true;} \\
&r: \text{pause}
\end{align*}
\]

\[
\begin{align*}
&q: \text{pause;} \\
&\text{if(!b) } c = \text{true;} \\
&a = \text{true;} \\
&s: \text{pause}
\end{align*}
\]

- 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)

\[ \implies \text{directly defines a simulator} \]

- **main idea**
  - introduce value \( \perp \): means ‘not yet known’
  - initially: all inputs known, all outputs unknown
  - estimate:
    - \( D_{\text{must}} \): set of all actions that must be fired
    - \( D_{\text{can}} \): 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

\[\Rightarrow \] 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!!**
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** \((\gamma, \alpha)\)
  - trigger condition \(\gamma\) with atomic action \(\alpha\)
  - \((\gamma, \alpha)\) is enabled, if \(\gamma\) holds
- **guarded actions reduce languages to their MoC**
  - recall MoC: when, why, which action is executed
  - \(\gamma\) is the reason (why?) for executing \(\alpha\) (which?)
  - ‘when’ is defined as:
    - synchronous MoC: execute **all enabled** actions
    - asynchronous MoC: execute **some enabled** actions
Translation to Guarded Actions

- idea: determine condition \( \gamma \) for each action \( \alpha \)
- however: very difficult to do
  - distinction between surface and depth required
  - modular translation very difficult due to reincarnations etc.

\( \Rightarrow \) 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:

\[
\begin{array}{c|c|c|c|}
\land & \bot & 0 & 1 \\
\hline
\bot & \bot & 0 & \bot \\
0 & 0 & 0 & 0 \\
1 & \bot & 0 & 1 \\
\end{array}
\quad
\begin{array}{c|c|c|c|}
\lor & \bot & 0 & 1 \\
\hline
\bot & \bot & \bot & 1 \\
0 & \bot & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{c|c|c|}
\times & \neg x \\
\hline
\bot & \bot \\
0 & 1 \\
1 & 0 \\
\end{array}
\]

- 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

\[
\begin{align*}
    o_1 \land \neg o_2 & \Rightarrow o_1 \\
    1 & \Rightarrow o_2
\end{align*}
\]

- causality analysis:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_1</td>
<td>⊥</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>o_2</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[\leadsto\text{ program is causal}\]
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 ($x \lor \neg x$) cannot be proved
  - all proof must be constructive

- **three-valued logic**
  - models progress of micro step execution
  - $\bot$: means not yet known
Hardware Synthesis

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

\[ \xrightarrow{\sim} \text{compute equations with carrier variable } x': \]

\[
x = \begin{cases} 
\text{case} \\
\gamma_1 : \tau_1; \\
\vdots \\
\gamma_p : \tau_p; \\
\text{else } x' 
\end{cases}
\]

\[
\text{next}(x') = \begin{cases} 
\text{case} \\
\chi_1 : \pi_1; \\
\vdots \\
\chi_q : \pi_q; \\
\text{else } \text{Default}(x) 
\end{cases}
\]

\[ \xrightarrow{\sim} x' \text{ stores delayed assignments of previous step} \]
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
  - potential exponential growth of code, but much faster
Synthesis of Parallel Software

- **multithreaded software is asynchronous!**
- \( \Rightarrow \textbf{translate guarded actions to DPN} \)
  - one node \( P_x \) for each output/state variable \( x \)
  - node \( P_x \) computes the value of \( x \) in each macro step
- \( \Rightarrow \textbf{static DPN:} \)
  - 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 $z = x \land y$

  $\Rightarrow$ **introduce new value $\square$ for analysis**
  - $\square$ is a placeholder for a concrete, but unwanted value
  - $\square$ will not be computed and also not communicated

- **compiler optimization:**
  - compute for every $(\gamma, \alpha)$ a condition $\beta$,
    such that $(\gamma \land \beta, \alpha)$ 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

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 :: $A$)</td>
<td>(b :: $B$)</td>
<td>(c :: $C$)</td>
<td>[b]</td>
</tr>
<tr>
<td>(0 :: $A$)</td>
<td>(b :: $B$)</td>
<td>(c :: $C$)</td>
<td>[c]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 :: $A$)</td>
<td>(b :: $B$)</td>
<td>C</td>
<td>[b]</td>
</tr>
<tr>
<td>(0 :: $A$)</td>
<td>B</td>
<td>(c :: $C$)</td>
<td>[c]</td>
</tr>
</tbody>
</table>

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

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

$\Rightarrow$ “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

\[
\begin{array}{cccc}
  x_1 & x_2 & x_3 & y \\
  (1 :: A) & (0 :: B) & C & [1] \\
  A & (1 :: B) & (0 :: C) & [1] \\
  (0 :: A) & B & (1 :: C) & [1]
\end{array}
\]

\[\Rightarrow\text{ desynchronization}\]

- generate DPN with sequential functions from synchronous guarded actions
- these DPNs can be run asynchronously
  \[\Rightarrow\text{ latency insensitive design/elastic circuits}\]
Boundedness for Parallel DPNs

- topology matrix yields the following solutions
  \[
  \begin{pmatrix}
  1 & -1 & 0 & 0 \\
  -1 & 1 & 0 & 0 \\
  0 & 1 & 0 & -1 \\
  0 & 0 & 1 & -3 \\
  0 & 0 & -1 & 3 \\
  \end{pmatrix}
  \]
  \[r = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 1 \end{pmatrix} \lambda\]

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

- buffer \((f_2, g_2)\) can overflow
- ‘backpressure’ edge from \(S_2\) to \(S_1\) required
- not necessary if \(T\) has \(S\)- and \(T\)-invariants
Model-based Design Using Averest

- **system behavior given by synchronous program**
  - precise formal semantics $\Rightarrow$ formal verification possible
  - deterministic/reproducible simulation
  - 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
The synchronous languages twelve years later.

_Introduction to Discrete Event Systems._

Hierarchical finite state machines with multiple concurrency models.

_Modeling Embedded Systems and SoCs._
The semantics of a simple language for parallel programming.  

Consistency in dataflow graphs.  

A framework for comparing models of computation.  

LCF considered as a programming language.  