



Models of computation and Languages

Slide -1 -

Premise

- Shrinking hardware costs, higher levels of integration allow more complex designs
- Designers' coding rate staying constant
- Higher-level languages the solution
 - Succinctly express complex systems

Diversity

- Why not just one “perfect” high-level language?
- Flexibility trades off analyzability
 - General-purpose languages (e.g., assembly) difficult to check or synthesize efficiently.
- Solution: Domain-specific languages

Domain-specific languages

- Language embodies methodology

Verilog

Model system and testbench

Multi-rate signal processing languages

Blocks with fixed I/O rates

Java's concurrency

Threads plus per-object locks to ensure atomic access

Types of Languages

- Hardware
 - Structural and procedural styles
 - Unbuffered “wire” communication
 - Discrete-event semantics
- Software
 - Procedural
 - Some concurrency
 - Memory
- Dataflow
 - Practical for signal processing
 - Concurrency + buffered communication
- Hybrid
 - Mixture of other ideas

Hardware Languages

- Goal: specify connected gates concisely
- Originally targeted at simulation
- Discrete event semantics skip idle portions
- Mixture of structural and procedural modeling

Hardware Languages

- Verilog
 - Structural and procedural modeling
 - Four-valued vectors
 - Gate and transistor primitives
 - Less flexible
 - Succinct
- VHDL
 - Structural and procedural modeling
 - Few built-in types; powerful type system
 - Fewer built-in features for hardware modeling
 - More flexible
 - Verbose

Hardware methodology

- Partition system into functional blocks
- FSMs, datapath, combinational logic
- Develop, test, and assemble
- Simulate to verify correctness
- Synthesize to generate netlist

Verilog

- Started in 1984 as input to event-driven simulator designed to beat gate-level simulators
- Netlist-like hierarchical structure
- Communicating concurrent processes
- Wires for structural communication,
- Regs for procedural communication

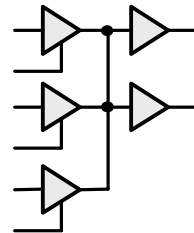
Verilog: Hardware communication

- Four-valued scalar or vector “wires”

```
wire alu_carry_out;  
wire [31:0] alu_operand;
```

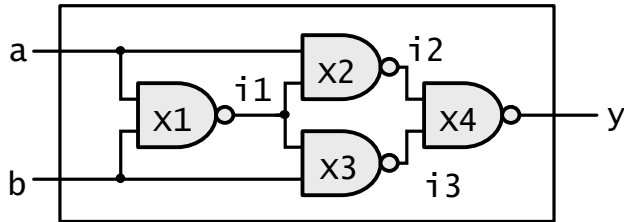
- X: unknown or conflict
- Z: undriven
- Multiple drivers and receivers
- Driven by primitive or continuous assignment

```
nand nand1(y2, a, b);  
assign y1 = a & b;
```



Verilog: Structure

```
module XOR(y, a, b);  
  output y;  
  input a, b;  
  
  NAND X1(i1, a, b),  
        X2(i2, a, i1),  
        X3(i3, b, i1),  
        X4(y, i2, i3);  
endmodule
```



Verilog: Software Communication

- Four-valued scalar or vector “register”

```
reg alu_carry_out;  
reg [31:0] alu_operand;
```

- Does not always correspond to a latch
- Actually shared memory
- Semantics are convenient for simulation
- Value set by procedural assignment:

```
always @(posedge clk)  
  count = count + 1;
```

Verilog: Procedural code

- Concurrently-running processes communicating through regs

```
reg [1:0] state; reg high, farm, start;
```

```
always @(posedge clk)
begin
  case (state)
  HG: begin
    high = GREEN; farm = RED; start = 0;
    if (car && long) begin
      start = 1; state = HY;
    end
  end
end
```

Verilog: Event control

- Wait for time

```
#10
a = 0;
```

- Wait for a change:

```
@(b or c);
a = b + c;
```

- Wait for an event:

```
@(posedge clk);
q = d;
```

Verilog: Blocking vs. Non-blocking

- Blocking assignments happen immediately

```
a = 5;  
c = a; // c now contains 5
```

- Non-blocking assignments happen at the end of the current instant

```
a <= 5;  
c <= a; // c gets a's old value
```

- Non-blocking good for flip-flops

VHDL

- Designed for everything from switch to board-level modeling and simulation
- Also has event-driven semantics
- Fewer digital-logic-specific constructs than Verilog
- More flexible language
 - Powerful type system
 - More access to event-driven machinery

Verilog and VHDL Compared

	Verilog	VHDL
Structure	●	●
Hierarchy	●	●
Separate interfaces		●
Concurrency	●	●
Switch-level modeling	●	○
Gate-level modeling	●	○
Dataflow modeling	●	●
Procedural modeling	●	●
Type system		●
Event access		●
Local variables		●
Shared memory	●	○
Wires	●	●
Resolution functions		●

Software Languages

- Goal: specify machine code concisely
- Sequential semantics:
 - Perform this operation
 - Change system state
- Raising abstraction: symbols, expressions, control-flow, functions, objects, templates, garbage collection

Software Languages

- C
 - Adds types, expressions, control, functions
- C++
 - Adds classes, inheritance, namespaces, templates, exceptions
- Java
 - Adds automatic garbage collection, threads
 - Removes bare pointers, multiple inheritance
- Real-Time Operating Systems
 - Add concurrency, timing control

Software methodology

- C
 - Divide into (recursive) functions
- C++
 - Divide into objects (data and methods)
- Java
 - Divide into objects, threads
- RTOS
 - Divide into processes, assign priorities

The C Language

- “Structured Assembly Language”
- Expressions with named variables, arrays

```
a = b + c[10];
```

- Control-flow (conditionals, loops)

```
for (i=0; i<10; i++) { ... }
```

- Recursive Functions

```
int fib(int x) {  
    return x = 0 ? 1 : fib(x-1) + fib(x-2);  
}
```

The C Language: Declarations

- Specifier + Declarator syntax for declarations

```
unsigned int      *a[10];
```

**Specifier: base type
and modifiers**

**Declarator: How to
reference the base
type (array, pointer,
function)**

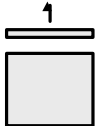
**Base types match
the processor's
natural ones**

The C Language: Storage Classes



Stack: Allocated and released when functions are called/return

Saves space, enables recursion



Heap: Allocated/freed by malloc(), free() in any order.

Flexible, slow, error-prone, can become fragmented



Static: Allocated when program is compiled, always present

Java: Simplified C++

- Simpler, higher-level C++-like language
- Standard type sizes fixed (e.g., int is 32 bits)
- No pointers: Object references only
- Automatic garbage collection
- No multiple inheritance except for interfaces:
method declarations without definitions

Java Threads

- Threads have direct language support
- `Object::wait()` causes a thread to suspend itself and add itself to the object's wait set
- `sleep()` suspends a thread for a specified time period
- `Object::notify()`, `notifyAll()` awakens one or all threads waiting on the object
- `yield()` forces a context switch

Java Locks/Semaphores

- Every Java object has a lock that at most one thread can acquire
- Synchronized statements or methods wait to acquire the lock before running
- Only locks out other synchronized code: programmer responsible for ensuring safety

```
public static void abs(int[] values) {  
    synchronized (values) {  
        for (int i = 0; i < values.length; i++)  
            if (values[i] < 0)  
                values[i] = -values[i];  
    }  
}
```

Java Thread Example

```
class OnePlace {
    Element value;

    public synchronized void
    write(Element e) {
        while (value != null) wait();
        value = e;
        notifyAll();
    }

    public synchronized Element read() {
        while (value == null) wait();
        Element e = value; value = null;
        notifyAll();
        return e;
    }
}
```

synchronized
acquires lock

wait suspends
thread

notifyAll
awakens all
waiting threads

Java: Thread Scheduling

- Scheduling algorithm vaguely defined
 - Made it easier to implement using existing thread packages
- Threads have priorities
- Lower-priority threads guaranteed to run when higher-priority threads are blocked
- No guarantee of fairness among equal-priority threads

Real-Time Operating Systems

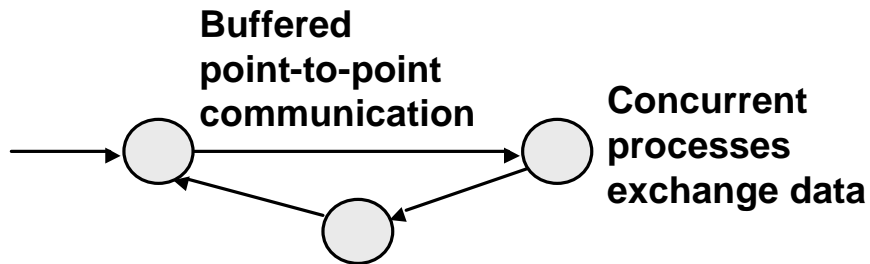
- Provides concurrency to sequential languages
- Idea: processes handle function, operating system handles timing
- Predictability, responsiveness main criteria

Software languages compared

	C	C++	Java	RTOS
Expressions	●	●	●	
Control-flow	●	●	●	
Recursive functions	●	●	●	
Exceptions	○	●	●	
Classes, Inheritance		●	●	
Multiple inheritance		●	○	
Operator Overloading		●		
Templates		●		
Namespaces		●	●	
Garbage collection			●	
Threads, Locks			●	●

Dataflow Languages

- Best for signal processing



Dataflow Languages

- Kahn Process Networks
 - Concurrently-running sequential processes
 - Blocking read, non-blocking write
 - Very flexible, hard to schedule
- Synchronous Dataflow
 - Restriction of Kahn Networks
 - Fixed communication
 - Easy to schedule

Dataflow methodology

- Kahn:
 - Write code for each process
 - Test by running
- SDF:
 - Assemble primitives: adders, downsamplers
 - Schedule
 - Generate code
 - Simulate

Kahn Process Networks

- Processes are concurrent C-like functions
- Communicate through blocking read, nonblocking write

```
/* Alternately copy u and v to w, printing each */
process f(in int u, in int v, out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(v);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```

Wait for next
input on port

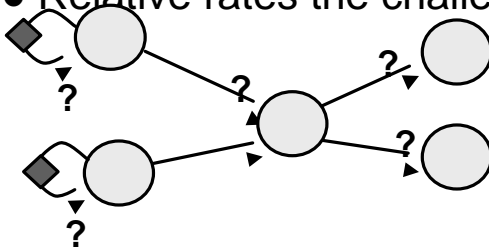
Send data on
given port

Kahn Networks: Determinacy

- Sequences of communicated data *does not* depend on relative process execution speeds
 - A process cannot check whether data is available before attempting a read
 - A process cannot wait for data on more than one port at a time
 - Therefore, order of reads, writes depend only on data, not its arrival time
 - Single process reads or writes each channel

Kahn Processes: Scheduling

- Relative rates the challenge

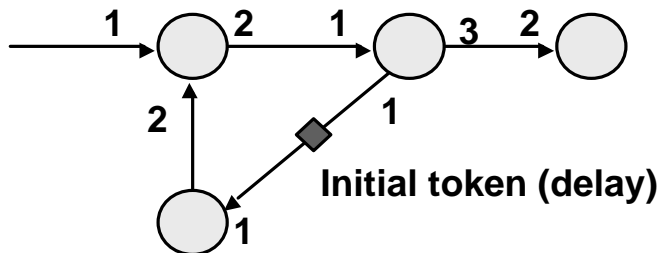


Which process should run next?

**One solution: Start with bounded buffers.
Increase the size of the smallest buffer when
buffer-full deadlock occurs.**

Synchronous Dataflow

- Each process has a firing rule:
 - Consumes and produces a fixed number of tokens every time
- Predictable communication: easy scheduling
- Well-suited for multi-rate signal processing
- A subset of Kahn Networks: deterministic



SDF Scheduling 1

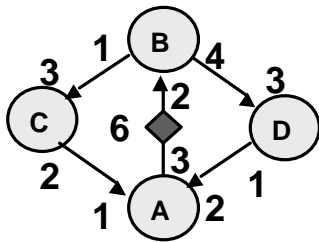
- Each arc imposes a rate constraint



- Solving the system answers how many times each actor fires per cycle
- Valid schedule: any one that fires actors this many times without underflow

SDF Scheduling 2

- Code generation produces nested loops with each block's code inlined
- Best code size comes from single-appearance schedule



(3B)C(4D)(2A)

**SAS: minimum
code size**

(3BD)BCA(2D)A

**Smaller
buffer
memory**

Dataflow languages compared

	Kahn	SDF
Concurrent	●	●
FIFO communication	●	●
Deterministic	●	●
Data dependent behavior	●	
Fixed rates		●
Statically schedulable		●

Summary of DF

- Advantages:
 - Easy to use
(graphical languages: Cossap, SPW, SystemStudio, Simulink, ...)
 - Powerful algorithms for
 - verification (fast behavioral simulation)
 - synthesis (scheduling and allocation)
 - Explicit concurrency
- Disadvantages:
 - Efficient synthesis only for restricted models
(no input or output choice)
 - Cannot describe reactive control
(blocking read)

Finite State Machines

- Typical domain of application: control-dominated systems
 - Control functions (automotive, UIs)
 - Protocols (telecom, computers, ...)
- Different communication mechanisms
 - Synchronous (classical FSMs, StateCharts, Esterel, ...)
 - Asynchronous (CCS, CSP, SDL, ...)
- Animated simulation
- SW and HW synthesis
- Extended with arithmetic operations
- Hierarchical states necessary for complex reactive system specification
- *Synchronous* graphical and textual languages

Hybrid Languages

- A mixture of ideas from other more “pure” languages
- Amenable to both hardware and software implementation

Hybrid Languages

- Esterel
 - Synchronous hardware model with software control-flow
- Polis
 - Finite state machine plus datapath for hardware/software implementation
- SDL
 - Buffered communicating finite-state machines for protocols in software
- SystemC
 - System modeling in C++, allowing refinement
- CoCentric™ System Studio
 - Dataflow plus Esterel-like synchrony

Hybrid Methodologies

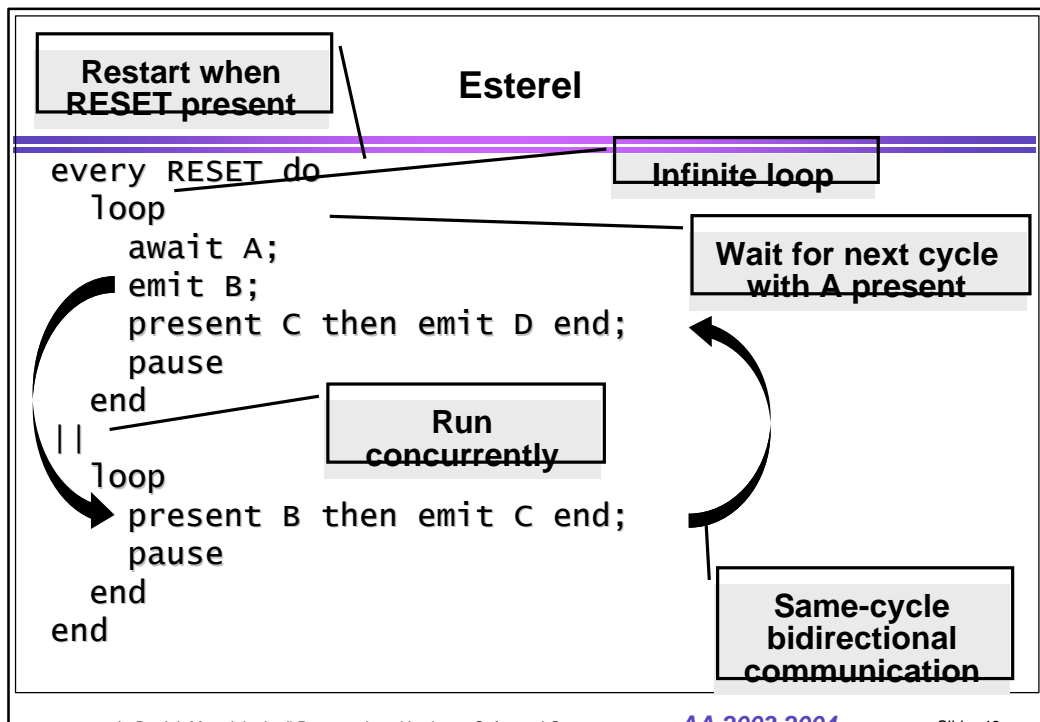
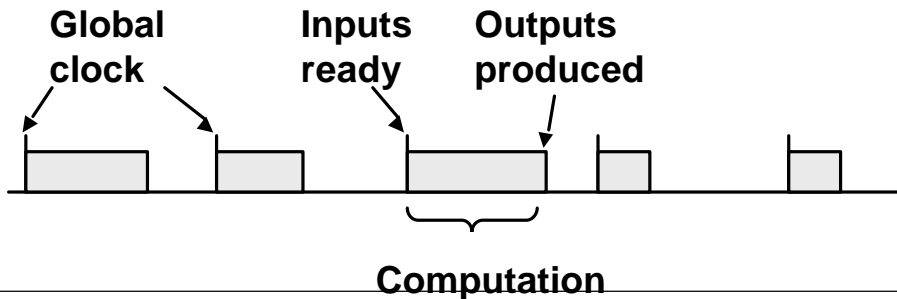
- Esterel
 - Divide into processes, behaviors
 - Use preemption
- Polis
 - Divide into small processes, dataflow
 - Partition: select hardware or software for each
 - Simulate or synthesize
- SDL
 - Divide into processes
 - Define channels, messages passed along each
 - Create FSM for each process

Hybrid Methodologies

- SystemC
 - Start with arbitrary C and refine
 - Divide into processes
 - Combine hierarchically
 - Simulate, Synthesize
- CoCentric™ System Studio
 - Assemble standard components
 - Add custom dataflow, control subsystems
 - Assemble hierarchically
 - Simulate, possibly embedded in another simulator

Esterel: Model of Time

- Like synchronous digital logic
 - Uses a global clock
- Precise control over which events appear in which clock cycles



Esterel Preemption

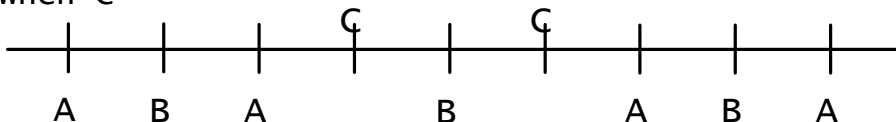
- Preempt the body before it runs
`abort
 body
when condition`
- Terminate the body after it runs
`weak abort
 body
when condition`
- Restart the body before it runs
`every condition do
 body
end`

Bodies may be concurrent

Esterel Suspend statement

- Strong preemption
- Does not terminate its body

```
suspend  
  loop  
    emit A; pause;  
    emit B; pause  
  end  
when C
```



Esterel Exceptions

- Exceptions a form of weak preemption
- Exit taken after peer threads have run
- Here, A and B are emitted in the second cycle

```
trap T in
```

```
  pause;  
  emit A
```

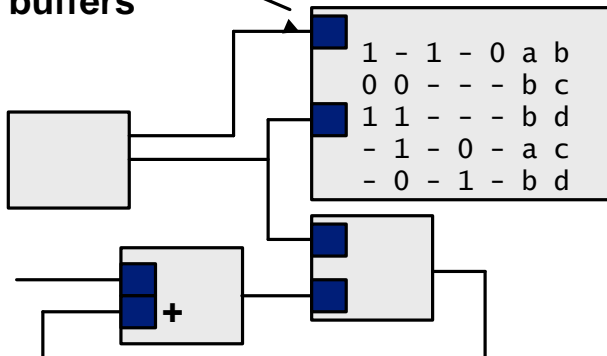
```
||
```

```
  pause;  
  exit T
```

```
handle T do  
  emit B  
end
```

Polis

Single-place input buffers



Reactive finite-state machines defined by tables

➤
Datapath elements

Polis communication

- Channels convey either values or events
- Only events cause CFSM transitions, but a CFSM can also read a value
- A CFSM consumes all its events after each transition

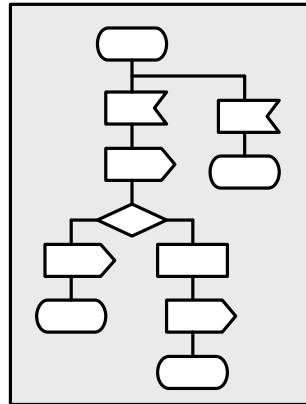
Polis Semantics

- Communication time is arbitrary
- CFSM computation time is non-zero, but arbitrary
- Events that arrive while a CFSM is transitioning are ignored
- The event in a valued event is read before its presence/absence, value is written first

SDL

- Concurrent FSMs, each with a single input buffer

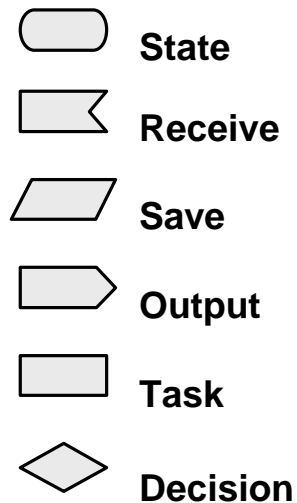
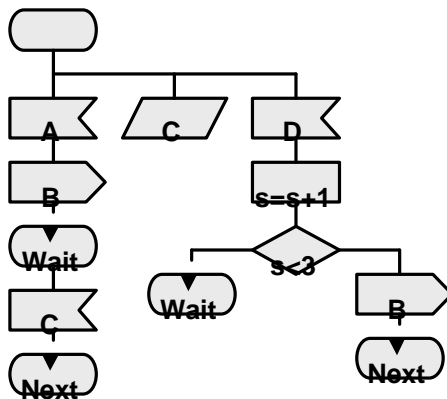
Finite-state machines defined using flowchart notation



[a b reset]

Communication channels define what signals they carry

SDL Symbols



SystemC

```
struct complex_mult : sc_module {
    sc_in<int> a, b;
    sc_in<int> c, d;
    sc_out<int> x, y;
    sc_in_clk clock;

    void do_mult() {
        for (;;) {
            x = a * c - b * d;
            wait();
            y = a * d + b * c;
            wait();
        }
    }

    SC_CTOR(complex_mult) {
        SC_THREAD(do_mult, clock.pos());
    }
};
```

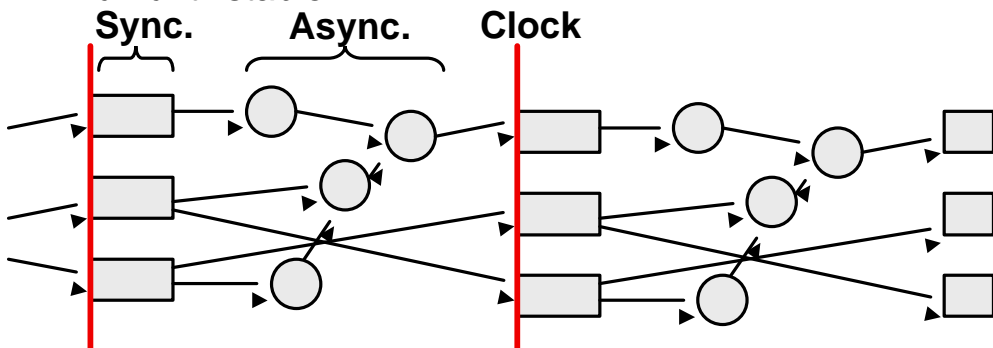
**Modules with
ports and
internal signals**

**Imperative code
with wait
statements**

**Instances of
processes, other
modules.**

SystemC Semantics

- Multiple synchronous domains
- Synchronous processes run when their clock occurs.
- Asynchronous processes react to output changes, run until stable

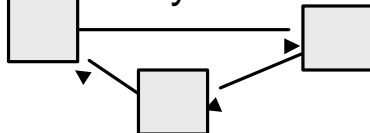


SystemC Libraries and Compiler

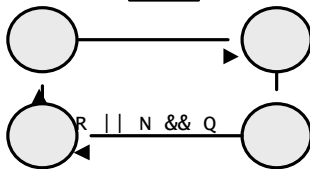
- SystemC libraries
 - C++ Class libraries & thread package
 - Allows SystemC models to be compiled and simulated using standard C++ compiler
 - Freely available at www.systemc.org
- CoCentric™ SystemC Compiler
 - Compiles SystemC models into optimized hardware
 - Commercial product from Synopsys

CoCentric™ System Studio

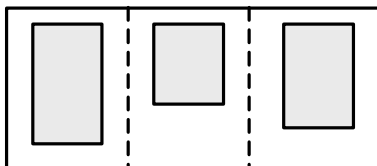
- Hierarchy of dataflow and FSM models



**Kahn-like
Dataflow**



OR models: FSMs



**AND models:
Esterel-like
synchronous
concurrency**

CoCentric™ System Studio

- AND models
 - Concurrent with Esterel-like semantics
 - Signals read after they are written
- OR models
 - Finite-state machines
 - Weak transitions: tested after the state's action is performed
 - Strong transitions: tested before the action
 - Immediate transitions: tested when the state is entered. Disables the action if true

CoCentric System Studio: Dataflow

- Fixed or variable rate
- Static and dynamic scheduling
- “Prim” models describe Kahn-like dataflow processes in a C++ subset
- CCSS attempts to determine static communication patterns

```
prim_model adder
{
  type_param T = int;
  port in T In1;
  port in T In2;
  port out T Sum;
  main_action
  {
    read(In1);
    read(In2);
    Sum = In1 + In2;
    write(Sum);
  }
}
```

Hybrid Languages Compared

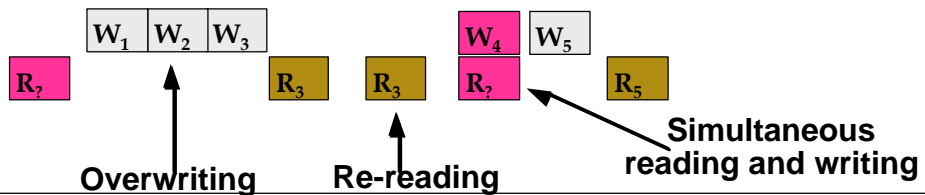
	Esterel	Polis	SDL	SystemC	CCSS
Concurrent	●	●	●	●	●
Hierarchy	●	●	●	●	●
Preemption	●			●	●
Deterministic	●			○	●
Synchronous comm.	●			●	●
Buffered comm.		●	●	●	●
FIFO communication			●		●
Procedural	●	○	○	●	○
Finite-state machines	●	●	●	○	●
Dataflow		●	●	●	●
Multi-rate dataflow					●
Software implement.	●	●	●	●	●
Hardware implement.	●	●		●	●

Common Models of Communication

- Model of Computation generally defines:
 - Process specification mechanism (computation)
 - Communication and synchronization mechanism
- Separate communication and computation for
 - Re-use of processes
 - Communication refinement
 - Communication elimination (static scheduling)
- “Basic” communication mechanisms :
 - Unsynchronized
 - Atomic read, write, read-modify-write
 - Unbounded FIFO buffer
 - Bounded FIFO buffer
 - Rendezvous

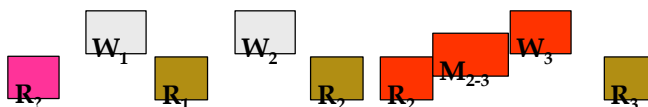
Unsynchronized

- Producer and consumer are not coordinated
- Used by
 - “Wild” multi-tasking software
 - Poorly synchronized hardware
- No guarantee of avoiding
 - Re-reading stale information
 - Overwriting
 - Simultaneous reading and writing (reading garbage...)



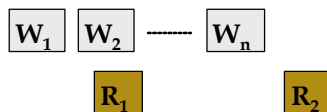
Atomic Read, Write, Read-Modify-Write

- Used by
 - “structured” multi-tasking software
 - well synchronized hardware
- Basis for implementation of RTOS and threading primitives:
 - semaphores,
 - java monitors,
 - other more complicated mechanisms
- Reading, writing and read-modify-write are atomic



Unbounded FIFO Buffer

- Point to point communication
- Avoids overwriting and re-reading (guarantees data safety and freshness)
- Used by
 - Kahn process networks
 - Dataflow networks (before static scheduling)
 - Petri Nets
 - SDL
- Hard to implement in memory-bounded embedded systems



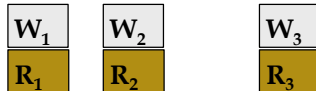
Bounded FIFO Buffer

- Same as before, but
 - Potentially more deadlocks
 - Implementable...
- Used by
 - Dataflow networks (after scheduling)
 - Synchronous FSMs (with potential overwriting)
- Imposes maximum difference (synchronic distance) between number of input and output events



Rendezvous

- Same as zero-sized FIFO buffer
- Used by CSP and CCS (FSM-based process algebras)
- One writing process and one reading process must be simultaneously “ready”
- Multi-way rendezvous possible
(But extremely hard to implement)



Basic Features of Communication

- Number of transmitters and receivers
 - Broadcast vs. Point-to-point
- Size of communication buffer
- Blocking read
 - Process cannot test for emptiness of input
 - Must wait for input to arrive before proceeding
- Blocking write
 - Process must wait for successful write before proceeding
- Single read (“destructive read”)
 - Result of each write can be read at most once

Comparing Communication Models

	Transm.	Receiv.	Buffer Size	Block. Read	Block. Write	Single Read
Unsynchronized	many	many	one	no	no	no
Read-Modify-write	many	many	one	yes	yes	no
Unbounded FIFO	one	one	unbound.	yes	no	yes
Bounded FIFO	one	one	bounded	no	maybe	yes
Single Rendezvous	one	one	one	yes	yes	yes
Multiple Rendezv.	many	many	one	no	no	yes

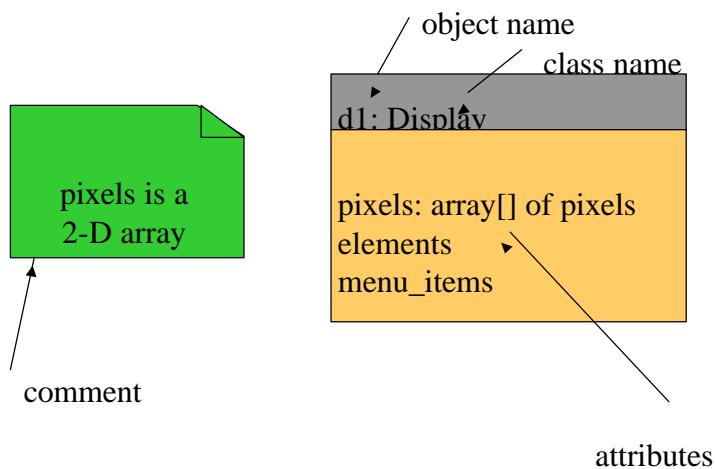
Summary of MOCs

- Mathematical notation for concurrent computation
- Essential for:
 - Simulation
 - Synthesis
 - Formal verification
- Model choice depends on
 - Class of applications
 - Required operations (synthesis, scheduling, ...)
- MOCs and languages can (and should) be mixed in real system designs

UML

- Developed by Booch et al.
- Goals:
 - object-oriented;
 - visual;
 - useful at many levels of abstraction;
 - usable for all aspects of design.

UML object

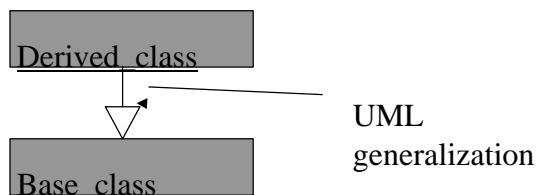


Relationships between objects and classes

- **Association**: objects communicate but one does not own the other.
- **Aggregation**: a complex object is made of several smaller objects.
- **Composition**: aggregation in which owner does not allow access to its components.
- **Generalization**: define one class in terms of another.

Class derivation

- May want to define one class in terms of another.
 - Derived class **inherits** attributes, operations of base class.

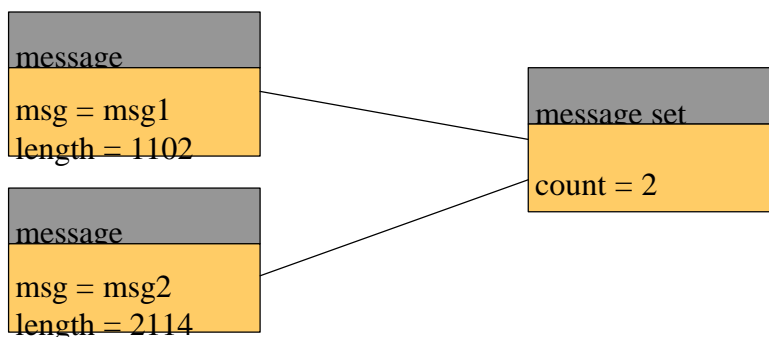


Links and associations

- **Link**: describes relationships between objects.
- **Association**: describes relationship between classes.

Link example

- Link defines the **contains** relationship:



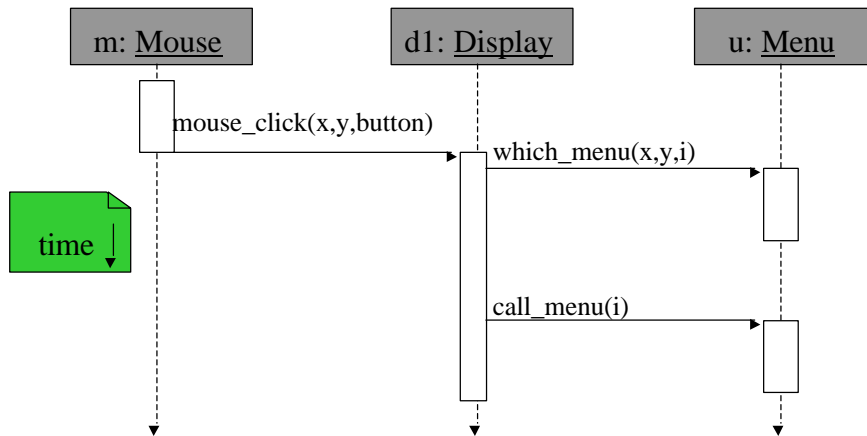
Stereotypes

- **Stereotype**: recurring combination of elements in an object or class.
- Example:
 - <<foo>>

Sequence diagram

- Shows sequence of operations over time.
- Relates behaviors of multiple objects.

Sequence diagram example



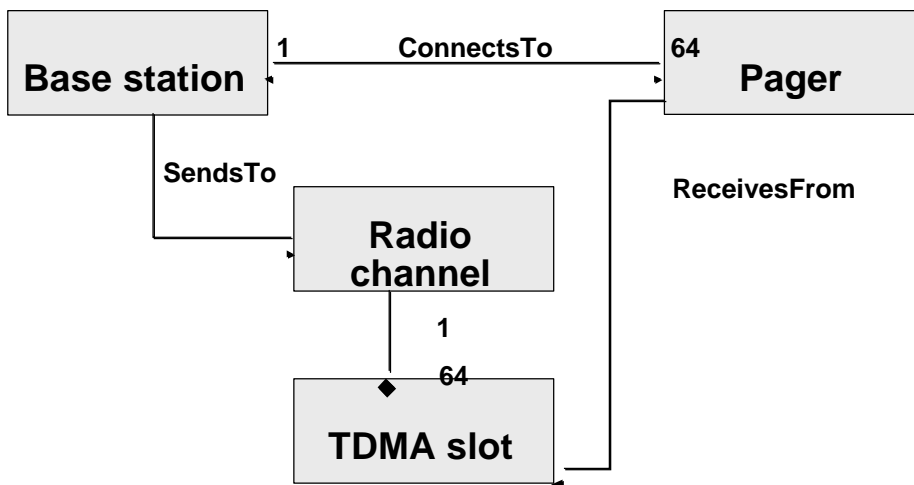
Summary

- Multiple models and languages are essential for high-level design
 - Managing complexity by abstraction
 - Formality ensures refinement correctness
 - Model choice depends on
 - Class of applications
 - Required operations (synthesis, scheduling, ...)
- Multiple MOCs can co-exist during all phases of design
 - Specification
 - Architectural mapping and simulation
 - Synthesis, code generation, scheduling
 - Detailed design and implementation
 - Co-simulation

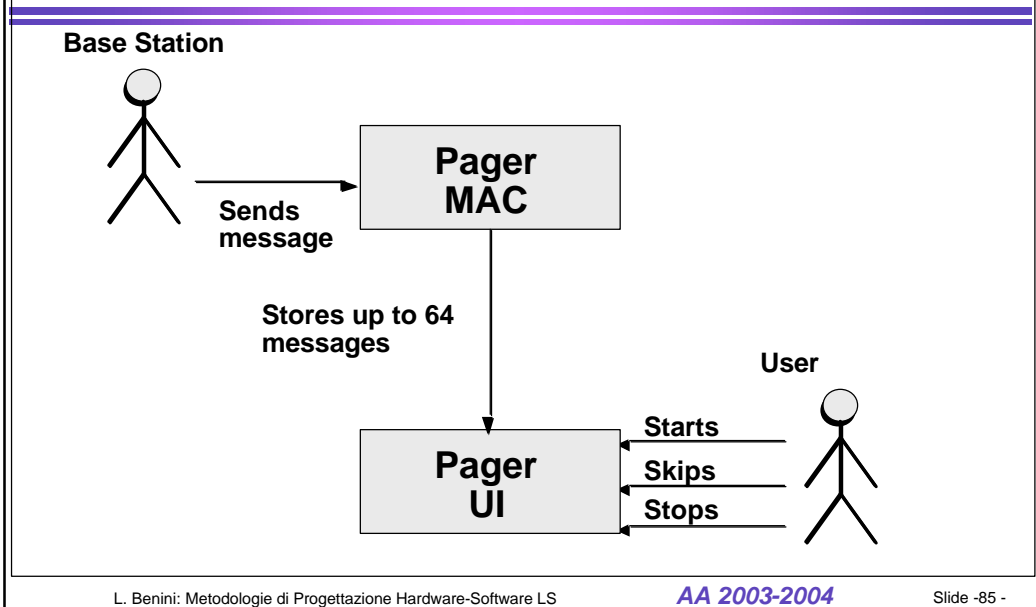
Example: Voice Mail Pager specification

- Receive Voice Mail messages
 - over a radio channel
 - using a fixed TDMA frame (64 VMPs per base station)
 - split into several packets (1024 bytes each)
- Play back Voice Mail messages
 - controlled by a simple answering machine-like User Interface
- Non-functional constraints
 - use legacy QCELP voice decoder written in assembler
 - output voice must be sampled at 8KHz

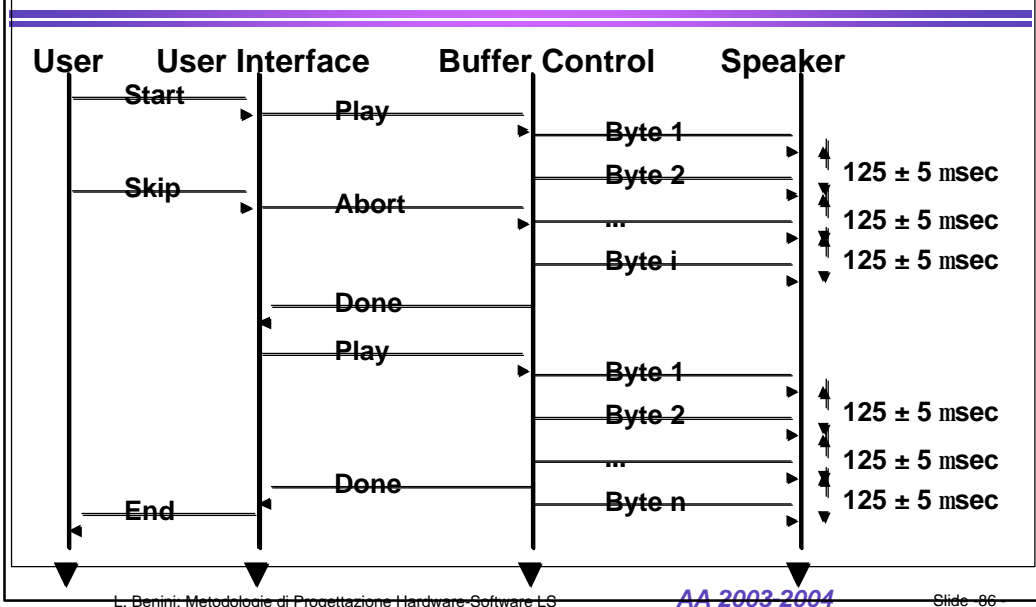
VMP radio Class Diagram



VMP maximum storage Use Case

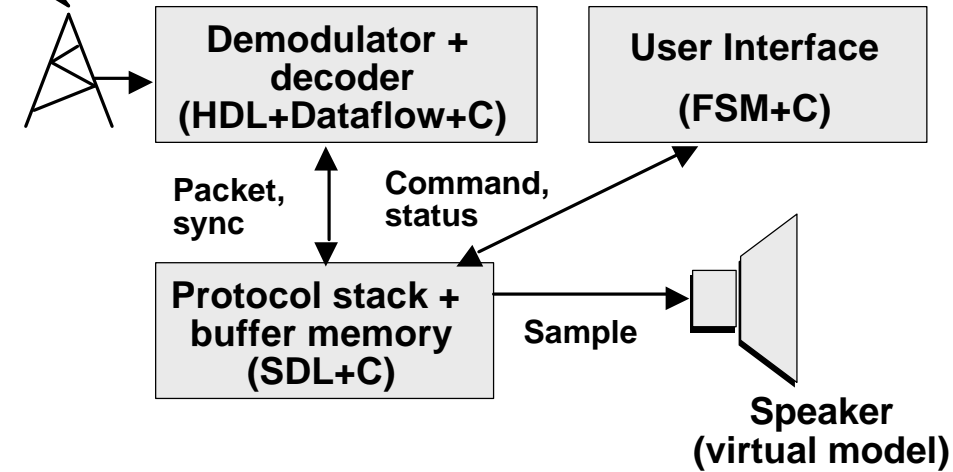


VMP play-skip Sequence Diagram

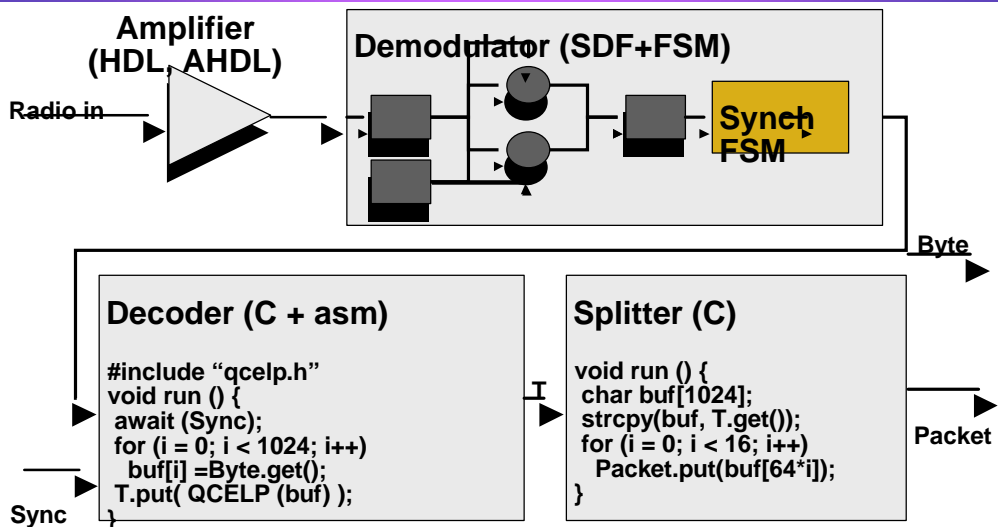


Functional executable specification

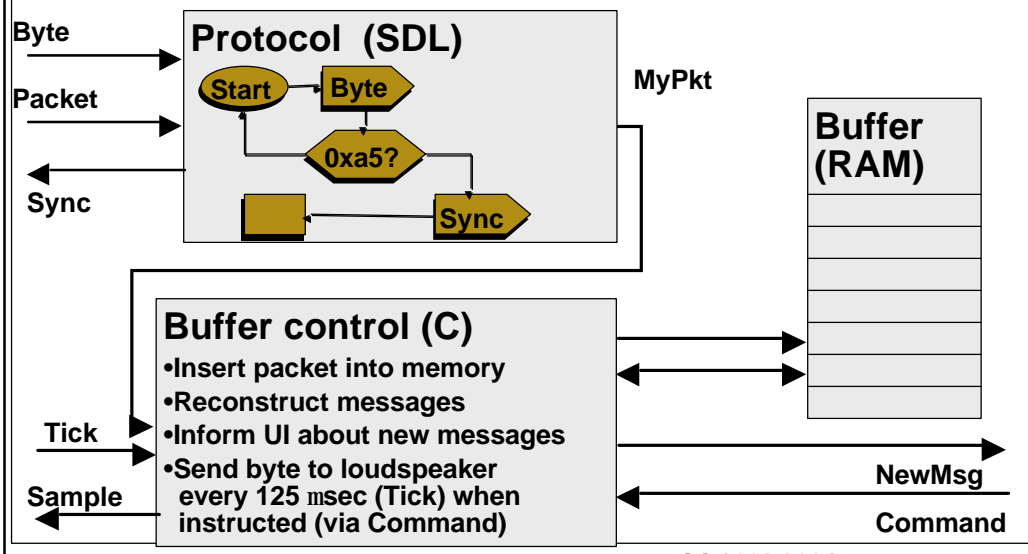
Channel + antenna
(analog model)



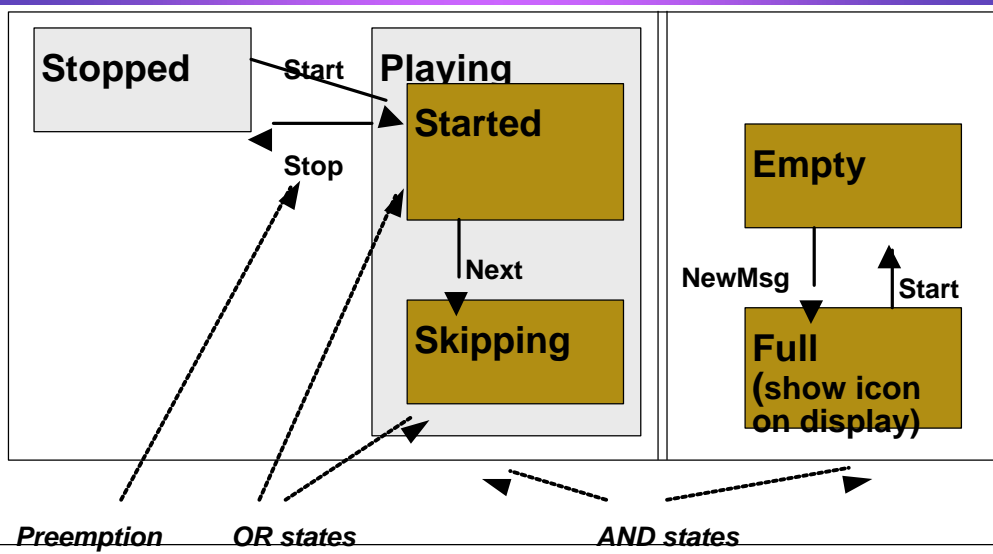
Demodulator and decoder



Protocol stack and buffer memory



User Interface (hierarchical FSM)



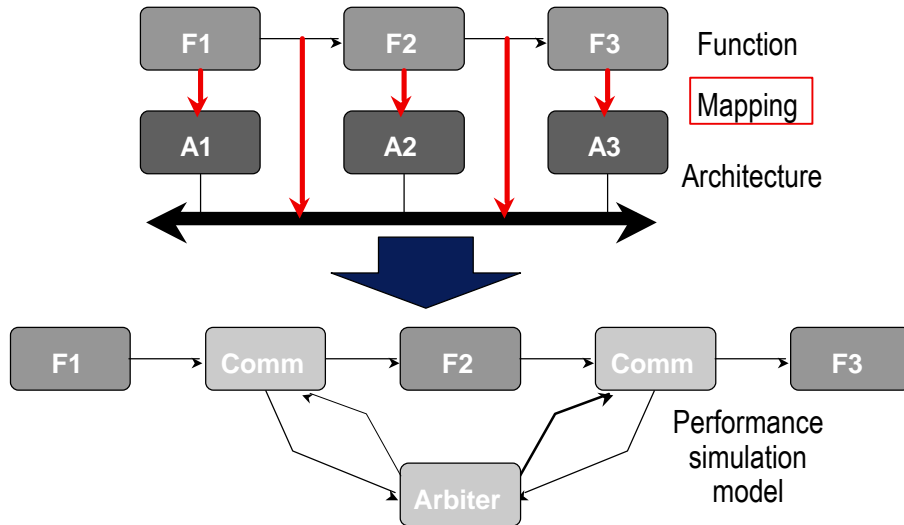
Heterogeneous modeling

- Use most appropriate language and MOC for each task
 - start with informal use cases in UML
 - refine using
 - differential equations (analog)
 - C and assembler code (legacy)
 - Static Dataflow network (DSP)
 - FSMs (protocols and UI)
 - Hardware Description Languages
 - simulate using Discrete Events (function and performance analysis)
 - implement using
 - synthesis (SDF, SDL, StateCharts)
 - manual coding (C, assembler, HDL)

Function and performance simulation

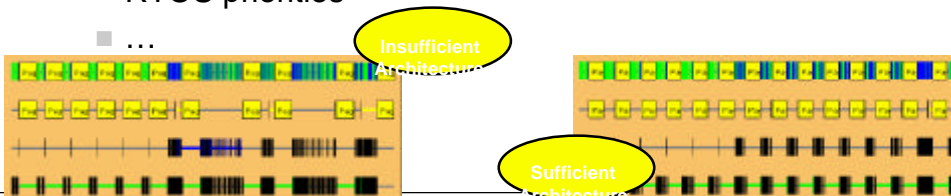
- Discrete Events is the “least common MOC”
- DE can be used to coordinate simulation of different MOCs within design (e.g., Ptolemy, SystemC 2.0, VCC)
- Functional simulation to verify functional use cases:
 - protocol with base station
 - maximum memory capacity
- Performance simulation to verify non-functional requirements:
 - timing constraints (sound quality)
 - CPU usage

Performance simulation by mapping



Performance simulation results

- Map functional blocks and communications onto architectural resources
- Verify architectural platform adequacy:
 - HW/SW partitioning
 - CPU speed
 - Bus bandwidth
 - RTOS priorities
 - ...



Implementation by mapping

- All this would be useless if we had to start from scratch with the next design step...
- Use mapping information to
 - synthesize functional block specifications, with a target-specific code generator
 - Esterel or StateChart code generator
 - Static Dataflow scheduler
 - RTL and logic synthesizer
 - synthesize resource schedulers
 - customize the RTOS (priority, scheduling policy, task creation, ...)
 - generate communication paths using
 - available resources (memories, busses, ...)
 - *refinement patterns* (interrupt, polling, DMA, ...)

Implementation by mapping

