# *Scaling the Abstraction Cliff: High-Level Languages for System Design*

Stephen A. Edwards
Synopsys, USA

Luciano Lavagno
University of Udine, Italy

## *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*

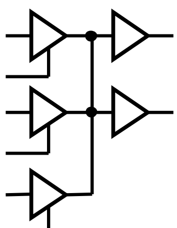
- Partition system into functional blocks
- FSMs, datapath, combinational logic
- Develop, test, and assemble
- Simulate to verify correctness
- Synthesize to generate netlist

## *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
```

## *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

## *VHDL: Entities and Architectures*

- **Entity: interface of an object**

```
entity mux2 is
  port(a,b,c: in Bit; d: out Bit);
end;
```

- **Architecture: implementation of an object**

```
architecture DF of mux2 is
begin
  d <= c ? a : b;
end DF;
```

## *VHDL: Architecture contents*

- **Structural, dataflow, and procedural styles:**

```
architecture ex of foo is
begin
  I1: Inverter port map(a, y);

  foo <= bar + baz;

  process begin
    count := count + 1;
    wait for 10ns;
  end
```

## *VHDL: Communication*

❖ **Processes communicate through resolved signals:**

```
architecture Structure of mux2 is
  signal i1, i2 : Bit;
```

❖ **Processes may also use local variables:**

```
process
 variable count := Bit_Vector (3 downto 0);
begin
 count := count + 1;
```

## *VHDL: The wait statement*

❖ **Wait for a change**

```
wait on A, B;
```

❖ **Wait for a condition**

```
wait on Clk until Clk = '1';
```

❖ **Wait with timeout**

```
wait for 10ns;
wait on Clk until Clk = '1' for 10ns;
```

## *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**

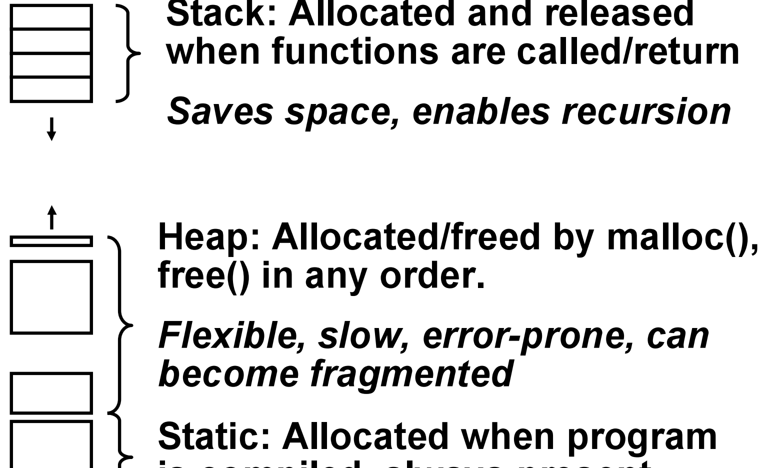
```
unsigned int   *a[10];
```

**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**

# C++: Classes

- **C with added structuring features**
- **Classes: Binding functions to data types**

```
class Shape {
  int x,y;
  void move(dx, dy) { x += dx; y += dy; }
};

Shape b;
b.move(10,20);
```

## C++: *Inheritance*

❖ **Inheritance: New types from existing ones**

```
class Rectangle : public Shape {
  int h, w;
  void resize(hh, ww) { h = hh; w = ww; }
};

Rectangle c;
c.resize(5,20);
c.move(10,20);
```

## C++: *Namespaces*

❖ **Grouping names to avoid collisions**

```
namespace Shape {
  class Rectangle { … };
  class Circle { … };

  int draw(Shape* s);
  void print(Shape* s);
}

Shape::Rectangle r;
```

## *C++: Templates*

❖ **Macros parameterized by types**

```
template <class T> void sort(T* ar)
{
  // …
  T tmp;
  tmp = ar[i];
  // …
}

int a[10];
sort(a);   // Creates sort<int>
```

## *C++: Exceptions*

❖ **Handle deeply-nested error conditions:**

```
class MyException {}; // Define exception

void bar()
{
  throw MyException;  // Throw exception
}

void foo() {
  try {
     bar();
  } catch (MyException e) { … } // Handle
}
```

## C++: Operator Overloading

❖ **Use expression-like syntax on new types**

```
class Complex {…};
Complex operator + (Complex &a, int b)
{
  // ...
}

Complex x, y;

x = y + 5;       // uses operator +
```

## C++: Standard Template Library

❖ **Library of polymorphic data types with iterators, simple searching algorithms**

- **vector: Variable-sized array**
- **list: Linked list**
- **map: Associative array**
- **queue: Variable-sized queue**
- **string: Variable-sized character strings with memory management**

## *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*

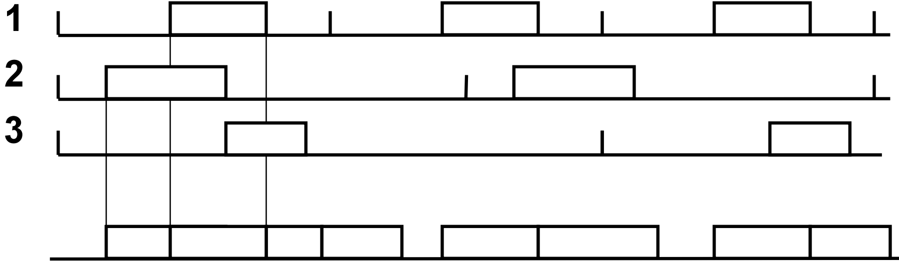
- Provides concurrency to sequential languages
- Idea: processes handle function, operating system handles timing
- Predictability, responsiveness main criteria

## *RTOS scheduling*

- ❖ Fixed-priority preemptive
- ❖ Sacrifices fairness to reduce context-switching overhead
- ❖ Meeting deadlines more important
- ❖ Process preempted when higher-priority process is activated
- ❖ Process otherwise runs until it suspends

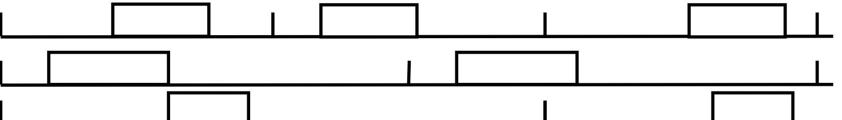
## *RTOS Scheduling*

- ❖ Highest-priority task always running
- ❖ Equal-priority tasks sometimes timesliced



## *Rate Monotonic Analysis*

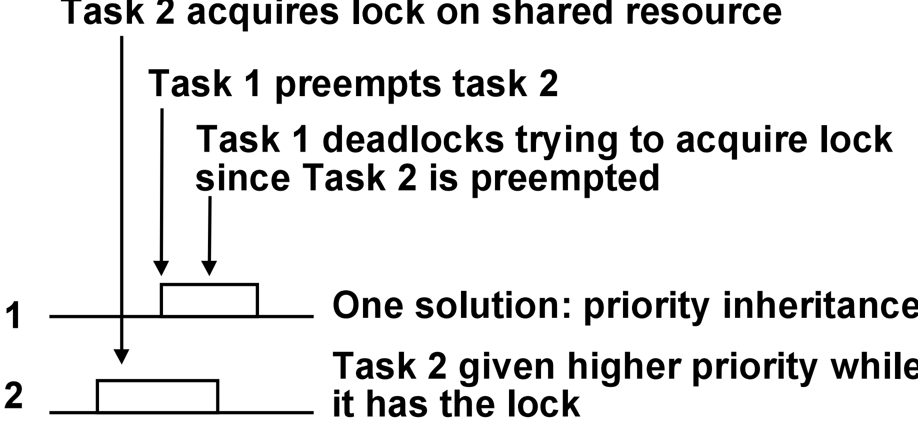
- Common priority assignment scheme
- System model:
  - Tasks invoked periodically
  - Each runs for some fraction of their period
  - Asynchronous: unrelated periods, phases



- Rate Monotonic Analysis:
  - Assign highest priorities to tasks with smallest periods

## *Priority Inversion*

- Deadlock arising when tasks compete for shared resources

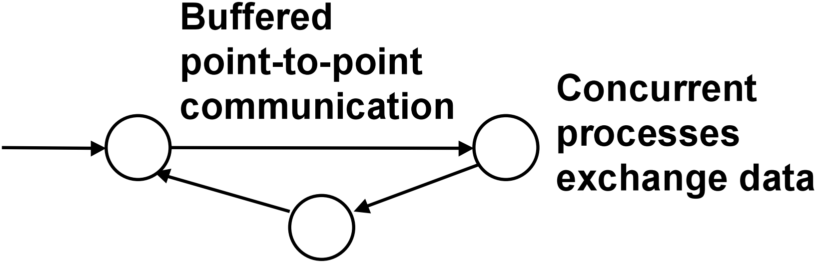


## *Software languages compared*

| | 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(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```

Wait for next input on port

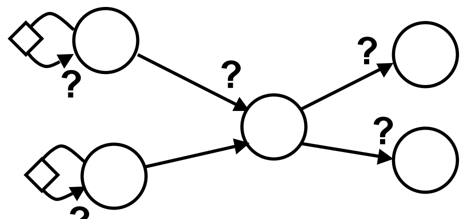
Send data on given port

## *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

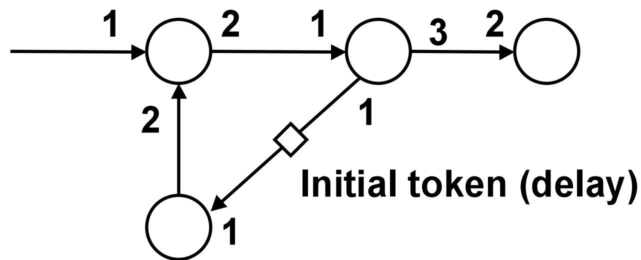



Which process should run next?

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

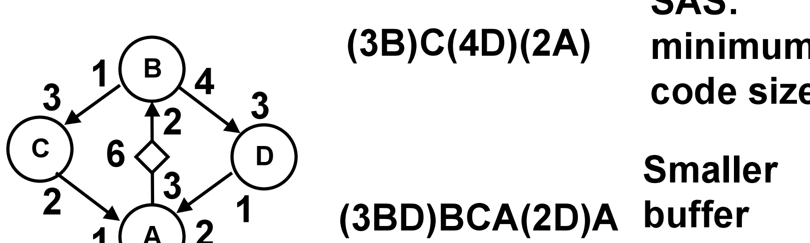
A 5 → 7 B

$5A = 7B$

- 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



## *Dataflow languages compared*

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

## *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*

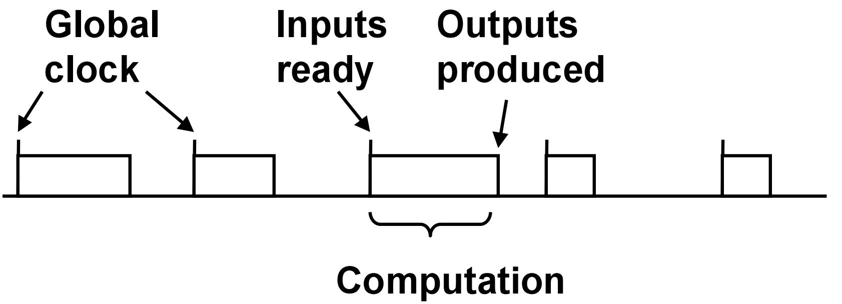
- Esterel
  - Divide into processes, behaviors
  - Use preemption
- Polis
  - Divide into small processes, dataflow
  - Partition: select hardware or software for each
  - Simulate or synthesize
- SDL
  - Divide into processes
  - Define channels, messages passed along each
  - Create FSM for each process

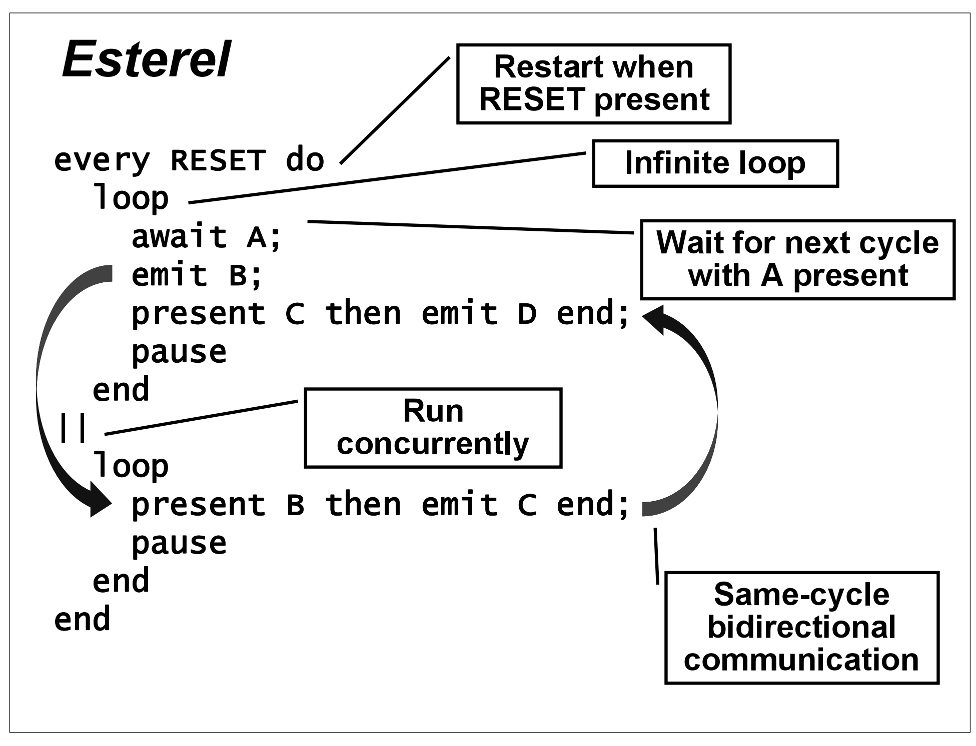
## *Hybrid Methodologies*

- 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**

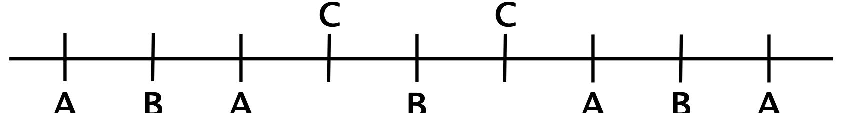
```
every condition do
   body
end
```

**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*

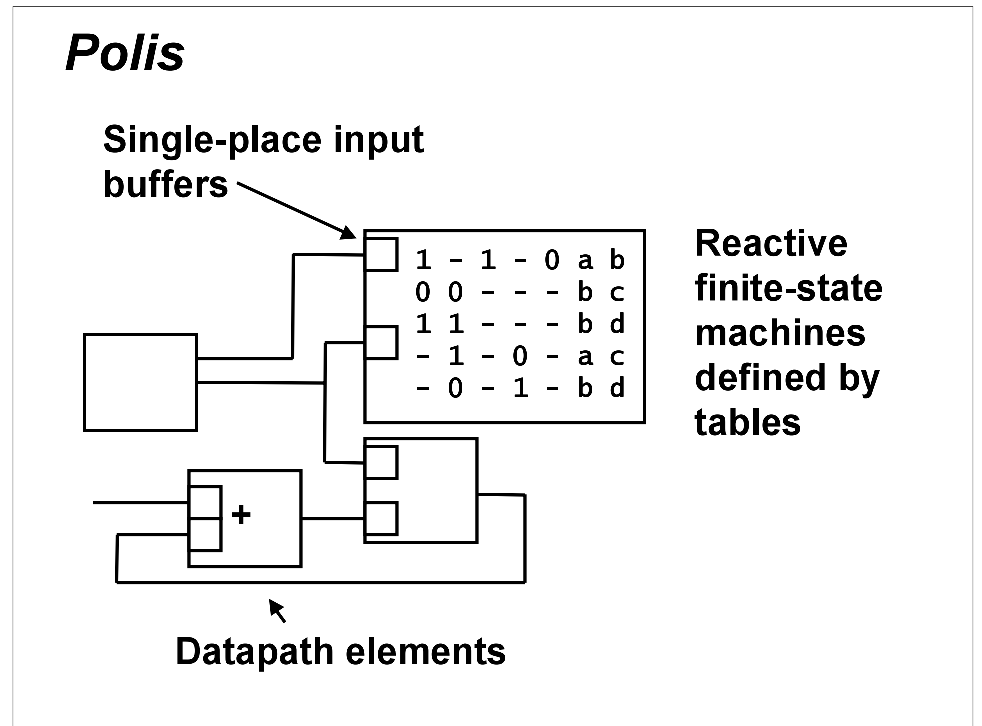
- Exceptions a form of weak preemption
- Exit taken after peer threads have run
- Here, A and B are emitted in the second cycle

```
trap T in

  pause;
  emit A
||
  pause;
  exit T

handle T do
  emit B
end
```

## *Polis*




## *Polis communication*

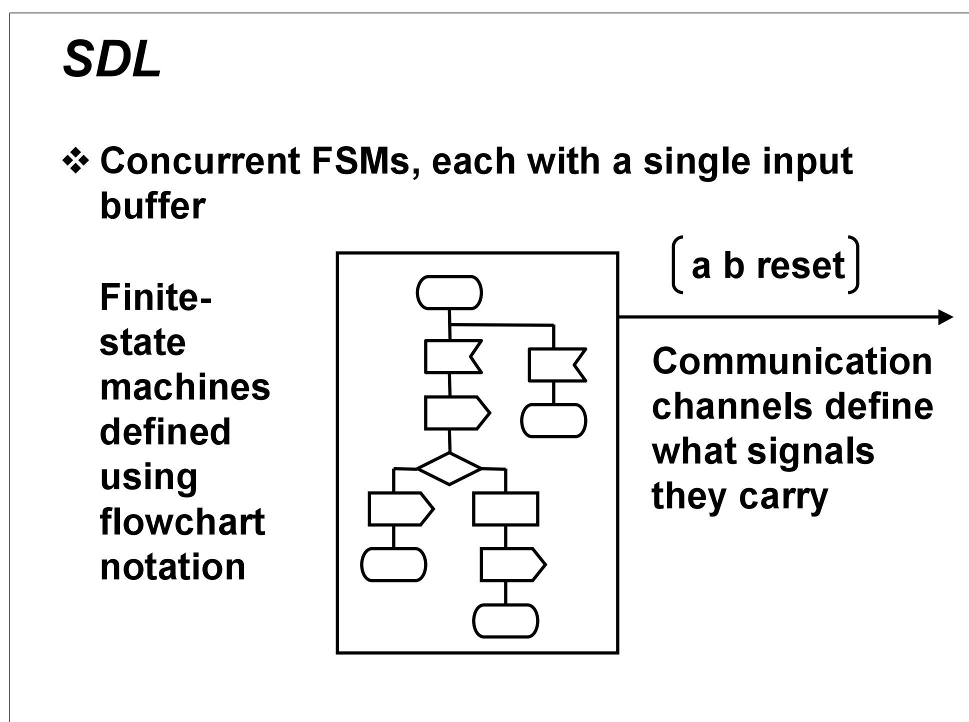
- Channels convey either values or events
- Only events cause CFSM transitions, but a CFSM can also read a value
- A CFSM consumes all its events after each transition

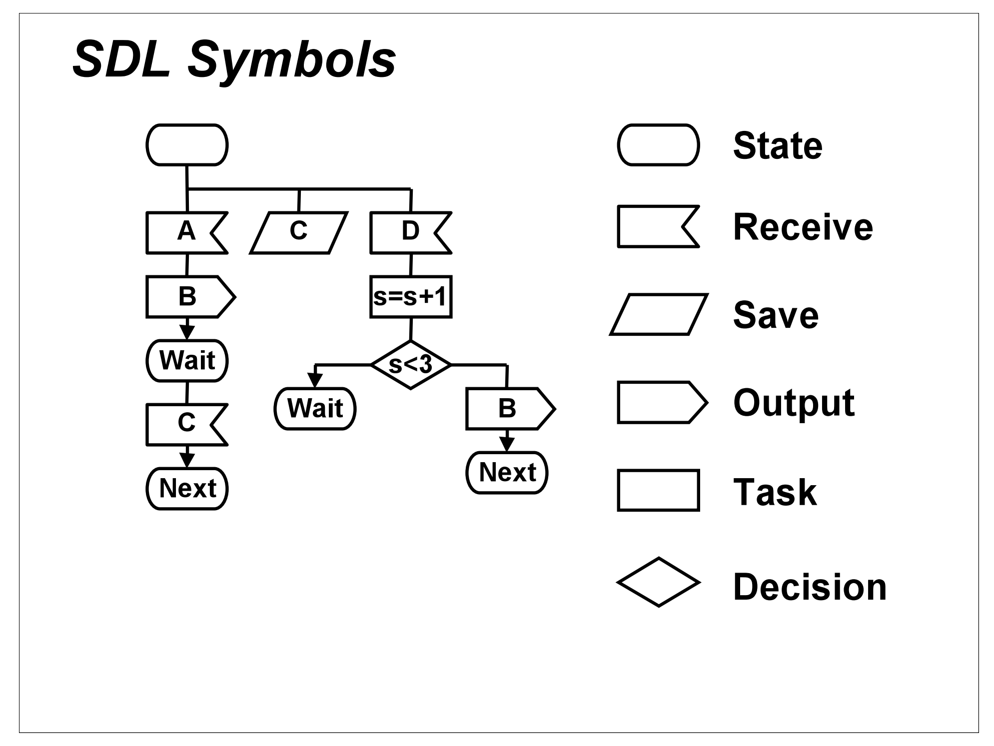
## *Polis Semantics*

- 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

## *Polis Synthesis*

- Software synthesis
  - Each CFSM becomes a process running under an RTOS
  - Buffers in shared memory
- Hardware synthesis
  - Each CFSM is a state machine
  - Transitions are taken in a single clock period
  - Inputs are latched


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
A
C
D
B
s=s+1
Wait
s<3
C
Wait
B
Next
Next
State
Receive
Save
Output
Task
Decision

## 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_CTHREAD(do_mult, clock.pos());
  }
};
```

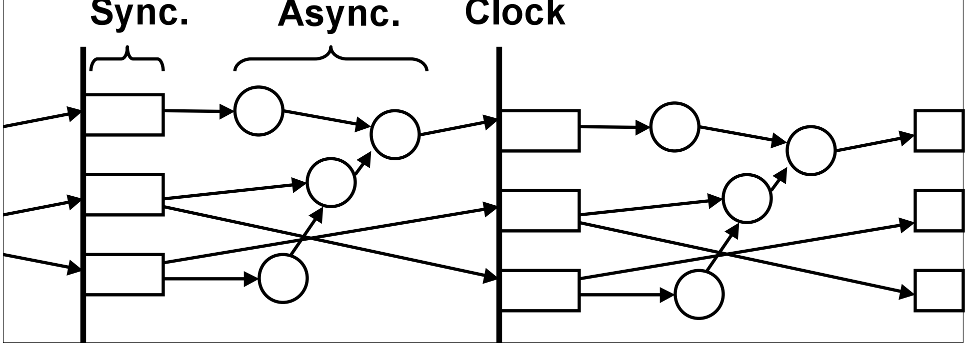
**Modules with ports and internal signals**

**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

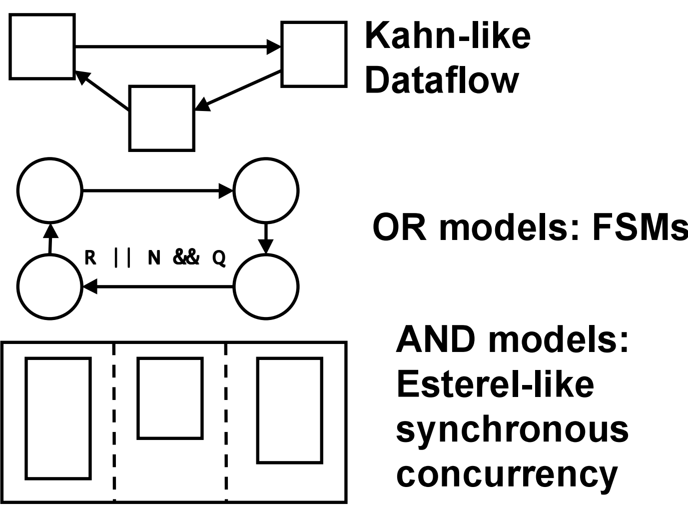


## *SystemC Libraries and Compiler*

- 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



## *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);
  }
}
```

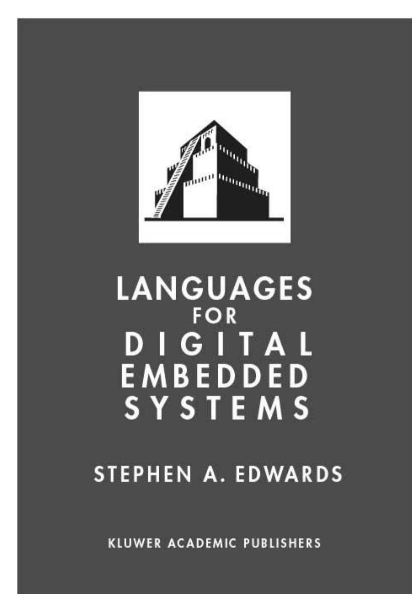
## *Hybrid Languages Compared*

| | 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. | ● | ● | | ● | ● |

## *Conclusions*

- Many types of languages
  - Each with its own strengths and weaknesses
  - None clearly “the best”
  - Each problem has its own best language
- Hardware languages focus on structure
  - Verilog, VHDL
- Software languages focus on sequencing
  - Assembly, C, C++, Java, RTOSes
- Dataflow languages focus on moving data
  - Kahn, SDF
- Others a mixture
  - Esterel, Polis, SDL, SystemC, System Studio

## *Shameless Plug*




**All of these languages are discussed in greater detail in**

**Stephen A. Edwards**
***Languages for Digital Embedded Systems***
**Kluwer 2000**