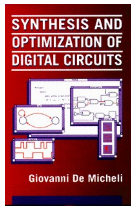
FSM-based Specification Formalisms

Giovanni De Micheli

Integrated Systems Centre

EPF Lausanne



SYNTHESIS AND OPTIMIZATION OF DIGITAL CIRCUITS

Giovanni De Micheli

Models of computation

- Data-flow oriented models
  - Focus on computation
  - Data-flow graphs and derivatives
- Control-flow oriented models
  - Focus on control
  - Based on *finite-state machine models*
- DF and CF model complementary aspects

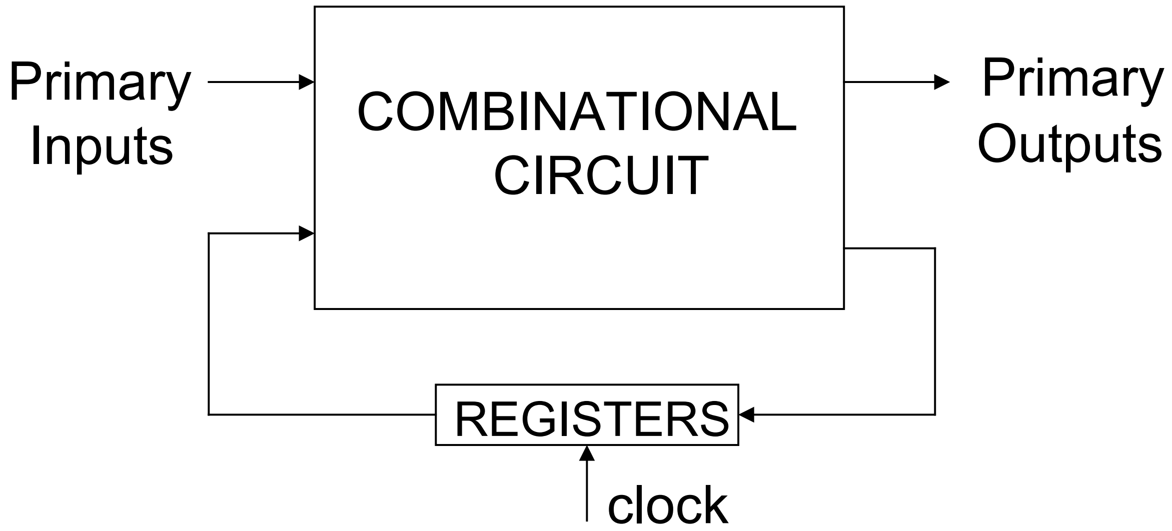
Module 1

- Objectives
  - FSM models
  - Languages with a synchronous semantics
  - Hierarchical FSMs
  - Expression-based formalisms

Formal FSM model

- A set of primary inputs patterns *X*
- A set of primary outputs patterns *Y*
- A set of states *S*
- A *state* transition function:
  - $\delta: X \times S \rightarrow S$
- An output function:
  - $\lambda : X \times S \rightarrow Y$ for *Mealy* models
  - $\lambda : S \rightarrow Y$ for *Moore* models

Finite-state machines

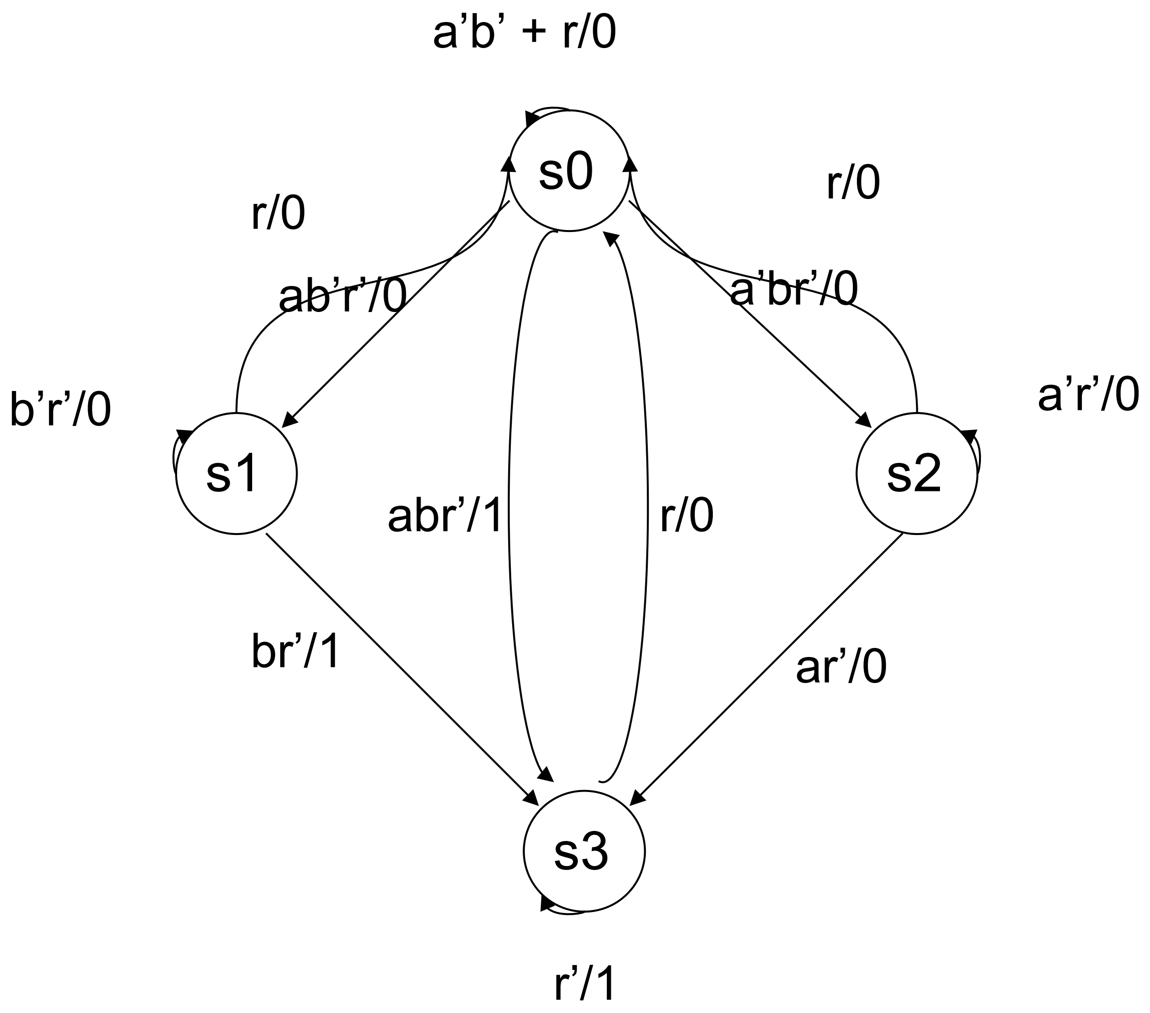


- A finite-state machine is an abstraction
- Computation takes no time
  - Outputs are available as soon as inputs are
- A finite-state machine implementation is a sequential circuit with a finite cycle-time

State diagrams

- Directed graph
  - Vertices = states
  - Edges = transitions
- Equivalent to state transition tables

Example



FSM-based models

- Synchronous languages:
  - Esterel, Argos, Lustre, SDL
- Graphical formalisms:
  - FSMs, hierarchical FSMs, concurrent FSMs
  - StateCharts
  - Program-state machines
  - SpecCharts

The synchronous approach

- Precise mathematical formalism
  - Strict semantics
  - FSM theoretical model
- Objectives
  - Support formal verification
  - Consider timing with behavior

Synchronous models

- Perfect synchrony hypothesis
  - Instantaneous response
  - Zero-delay computation
  - Outputs synchronous with inputs
- Discrete-time model
  - Sequence of *tics*
  - Environment driven
  - Inactivity between ticks

Event-controlled blocks (in Esterel)

- *Do* task *watching* event
  - Exit when event is present
- *Do* task *watching* event *timeout* task
  - Extension to time-out
- *Trap* and *handle*
  - Allow for exception handling

Example: speedometer design

```
module SPEED:
input SECOND, CM;
output SPEED: integer;
loop
    var NB_CM :=0: integer in
        do
            every CM do
                NB_CM := NB_CM + 1;
            end every
        watching SECOND;
        emit SPEED (NB_CM);
    end var
end loop
```

Example: jogging

```
do
    loop
         do RUN_SLOWLY watching 100 M;
         do
           every STEP do
                        JUMP || BREATHE
           end
         watching 15 S;
         RUN_FAST
      each LAP
watching 2 LAP
```

Example: jogging too much

```
trap HEART_ATTACK in
    do
            loop
                        do RUN_SLOWLY watching 100 M;
                        do
                                    every STEP do
                                                JUMP || BREATHE||CHECK_HEART
                                    end
                        watching 15 S;
                        RUN_FAST
        each LAP
    watching 2 LAP

handle HEART_ATTACK
    GO_TO_HOSPITAL
end
```

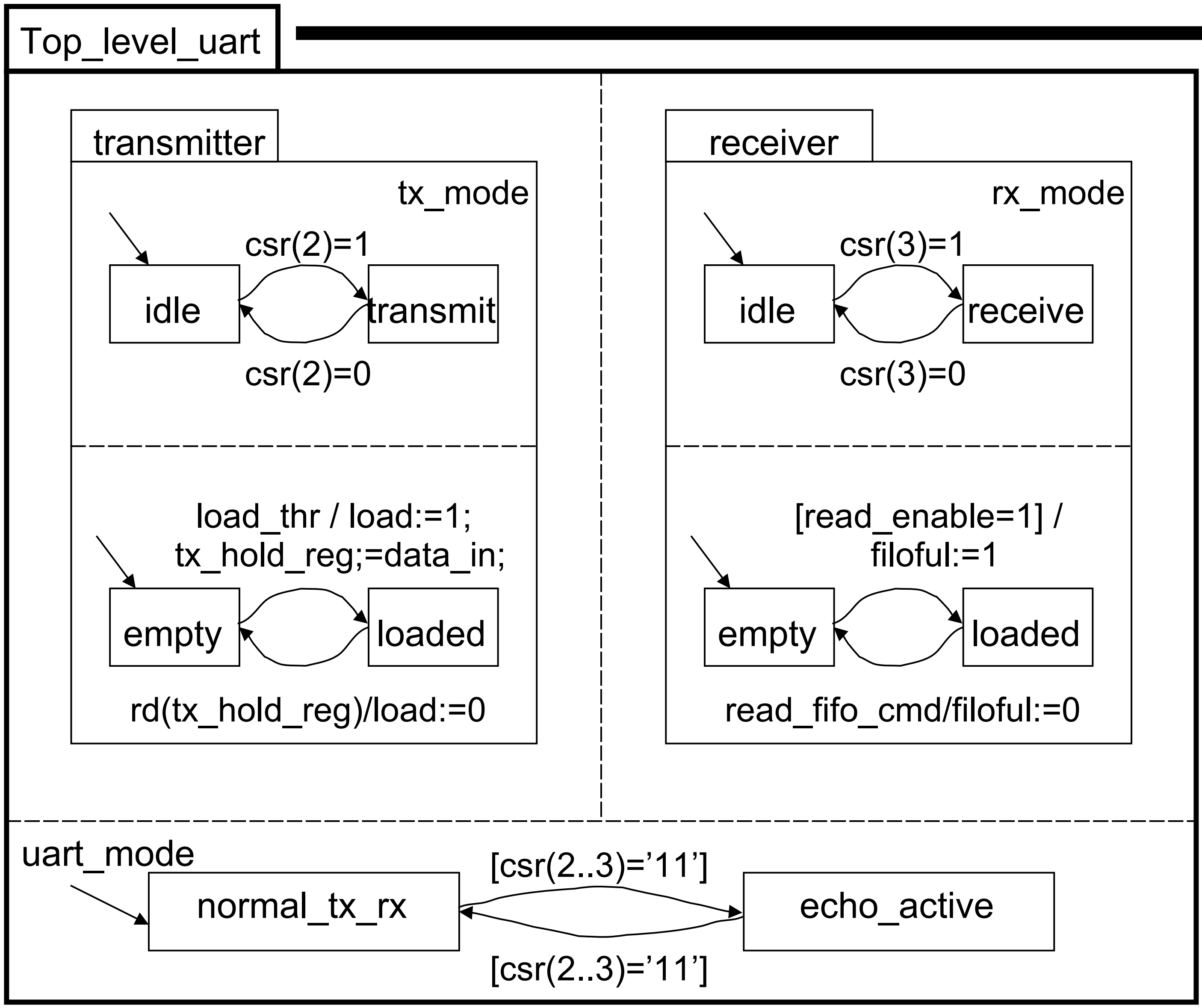
State Charts

- Proposed by Harel
- Graphic formalism to specify FSMs with:
  - Hierarchy
  - Concurrency
  - Communication
- Tools for simulation, animation and synthesis

State Charts

- States
- Transitions
- Hierarchy
  - OR (sequential) decomposition
    - State → a sequence of states
  - AND (concurrent) decomposition
    - State → a set of concurrent states

State charts



State Charts
Additional features

- State transitions across multiple levels
- Timeouts:
  - Notation on transition arcs denoting the max/min time in a given state
- Communication:
  - Broadcast mechanism based on event generation and reception
- History feature:
  - Keep track of visited states

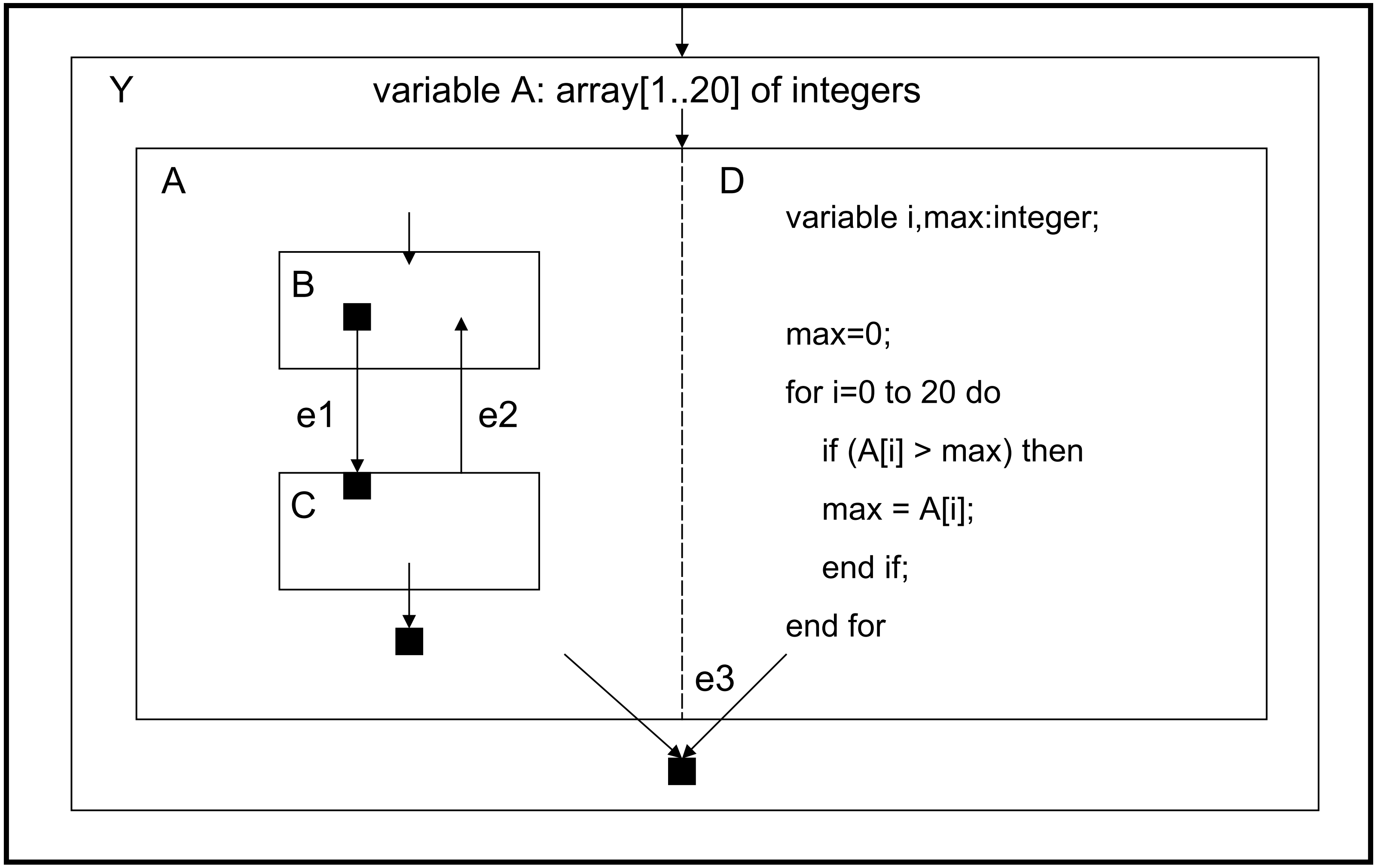
StateCharts

- **Advantages:**
  - Formal basis
  - Easy to learn
  - Support of hierarchy, concurrency and exceptions
    - Avoid exponential blow up of states
- **Disadvantages:**
  - No description of data-flow computation

Program State Machines

- Combining FSM formalism with program execution
- In each state a specific program is active
- Hierarchy:
  - Sequential states
  - Concurrent states
- In a hierarchical state, several programs may be active

Program state machine example



```
variable i,max:integer;

max=0;
for i=0 to 20 do
    if (A[i] > max) then
    max = A[i];
    end if;
end for
```

Program State Machine Transitions

- ***TOC - Transition on completion***
  - **Program terminates AND transition condition is true**
- ***TI - Transition immediate***
  - **Transition condition is true**

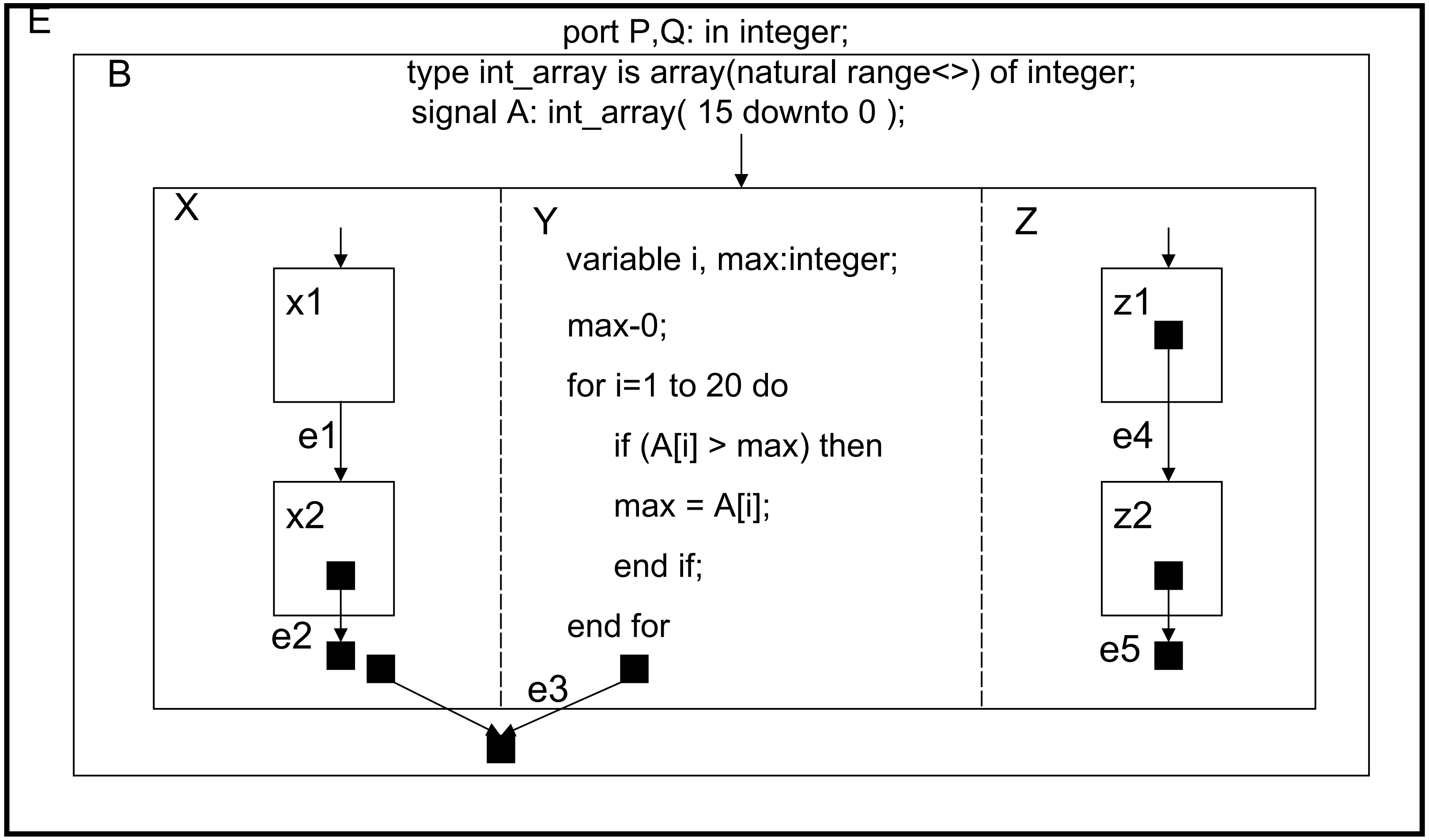
SpecCharts

- Based on Program State Machines
  - Introduced by Gajski et al.
- Extension of VHDL:
  - Compilable into VHDL for simulation and synthesis
  - Behavioral hierarchy
- Combining FSM and VHDL formalisms
  - Leaves of the hierarchy are VHDL models

State transitions

- Sequencing between sub-behaviors are controlled by transition arcs
- A transition arc is labeled by a triple:
  - (transition type, triggering event, next behavior)
- Transition types:
  - Transition on completion
  - Transition immediate
    - Timeout arcs

Example



- **TOC: e2, e3**
- **TI: e1**

SpecCharts semantics

- **Timing semantics similar to VHDL**
- **Synchronization:**
  - **Use *wait statement***
  - **Use TOC looping back to the top of the program**
- **Communication:**
  - **Using variables and signals**
  - **Message passing (send/receive)**

SpecCharts

- Language
- Graphic formalism
  - Similar to StateCharts
- Tools for analysis:
  - Area, expected performance
- Automatic conversion to VHDL

Expression-based formalisms

- Represent sequential behavior by expressions
- Advantages:
  - Symbolic manipulation
  - Translation into FSM models
- Disadvantages:
  - Loss of data-flow information

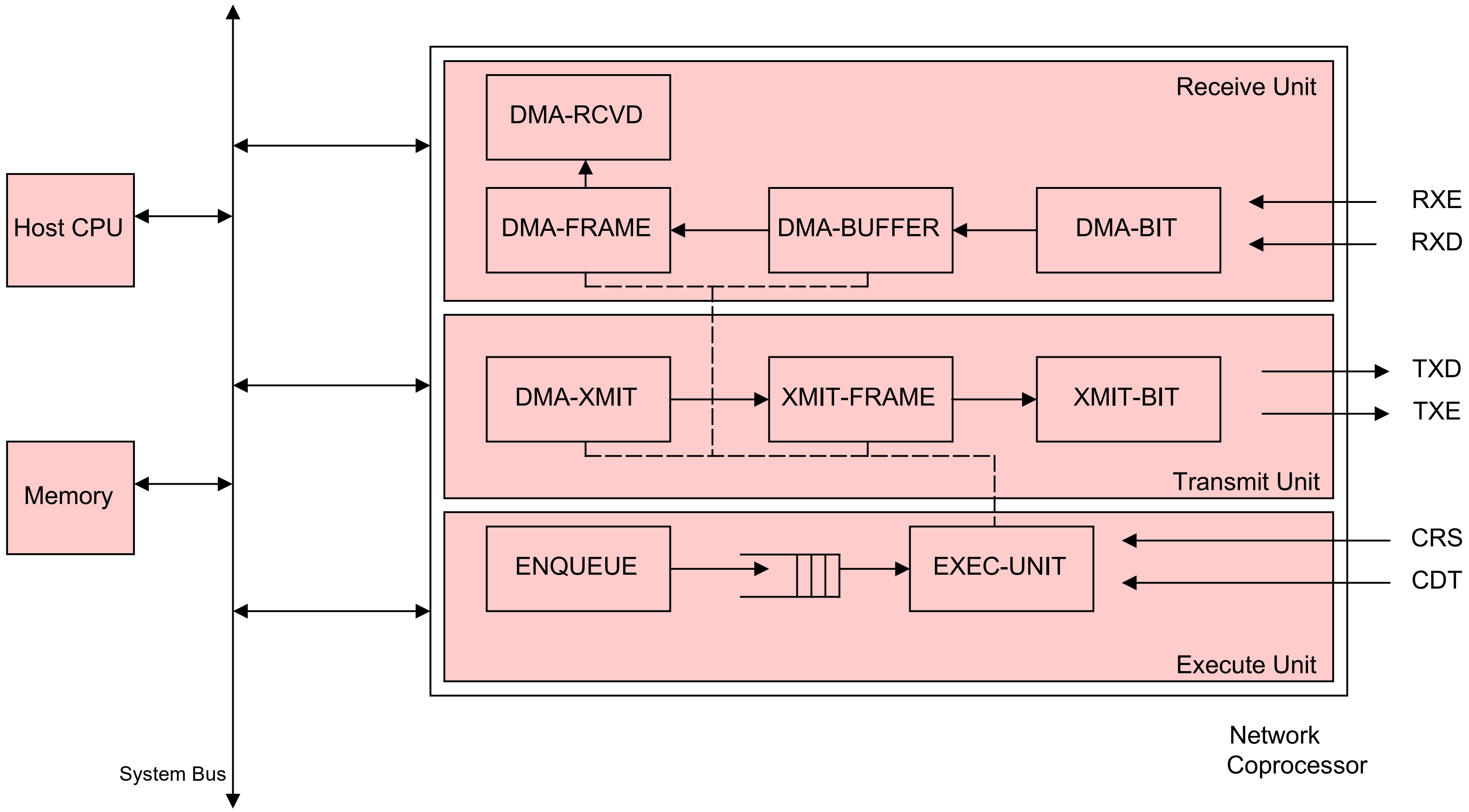
Regular expressions

- Model sequential/concurrent behavior
- Expressive power equivalent to FSMs
- Known techniques for synthesis and analysis
- Disadvantages:
  - No explicit way to express branching
  - No distinction between concurrent and alternative behavior

Control-flow expression formalism

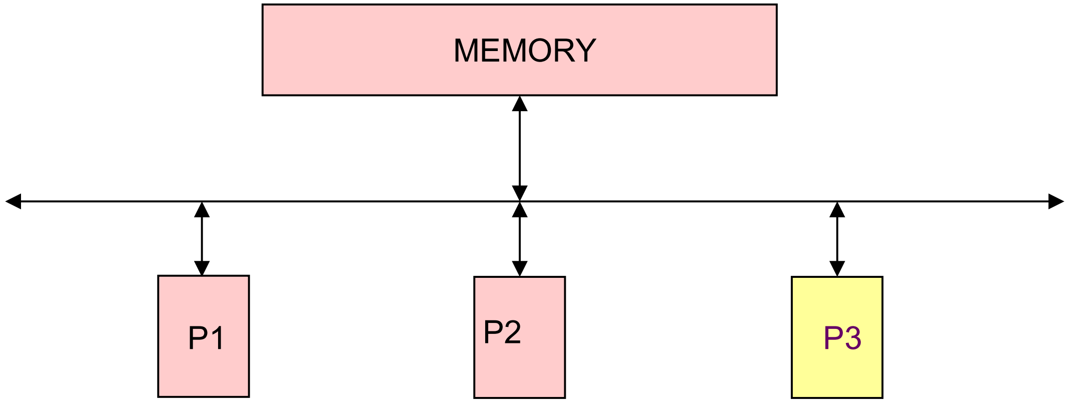
- Expressions capturing a *high-level view of control-flow* while abstracting data-flow information
- Expressions are extracted directly from HDL or programming language specifications
- *Cycle-based semantics* provides a formal interpretation of HDLs
- Based on the *algebra of synchronous processes* (Process Algebra)

Example of design problem Ethernet controller



- **Problem**
  - **Avoid bus conflicts**

Example of design problem Ethernet controller



```
always
begin
  write bus
  receive data
end
```

```
always
begin
  initialize
  wait ( tr ready )
  read bus
end
```

```
always
begin
  wait ( free bus )
  read bus
end
```

$$p = p1 \parallel p2 \parallel p3$$

$$p1 = [a.0]^{\omega}$$

$$p2 = [0.(c:0)^{*}.a]^{\omega}$$

$$p3 = [(x:0)^{*}.a]^{\omega}$$

Control-Flow Expressions

Composition	HDL	CFE
Sequential	**begin** P; Q **end**	$p \cdot q$
Parallel	**fork** P; Q **join**	$p \parallel q$

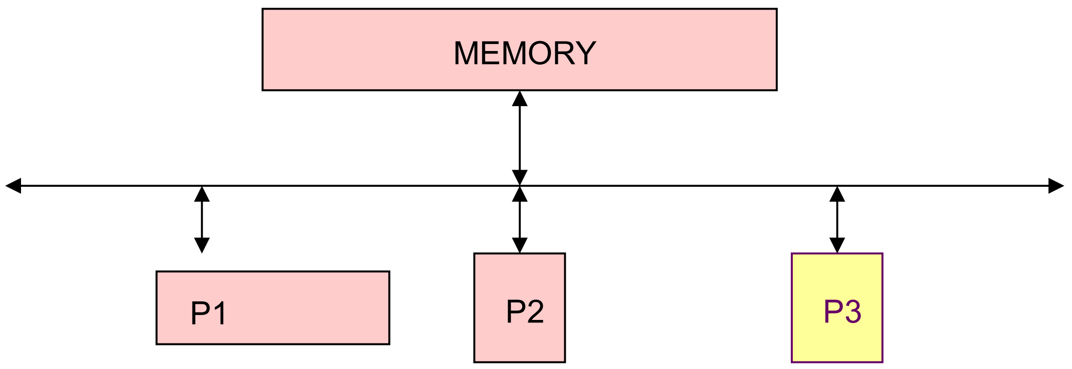
Alternative	**if** (c)
	P ;
	else
	Q ;
		$c : p + \overline{c} : q$

Loop	**while** (c) P ; **wait (!c)** P ;	$(c: p)^*$ $(c: 0)^* .p$
Infinite	**always** P ;	p^ω

Model properties

- Fully deterministic model.
  - Non-determinism captured by *decision* variables *affecting the clauses*
- Design space modeled by decision variables
  - An implementation is an assignment to decision variables over time
- Constraints expressible by CFEs
  - Timing, synchronization, resource usage
- AWAYS and NEVER sets
  - Set of actions that always/never execute simultaneously

Example
Control-Flow Expressions



```
always
begin
  write bus
  receive data
end
```

```
always
begin
  initialize
  wait ( tr ready )
  read bus
end
```

```
always
begin
  wait ( free bus )
  read bus
end
```

$p = p1 \parallel p2 \parallel p3$

$p1 = [a.0]^{\omega}$

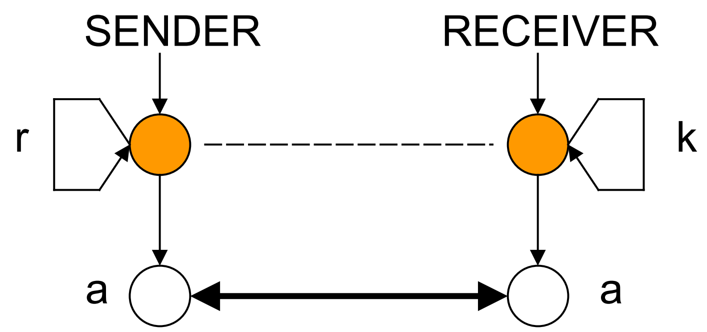
$p2 = [0.(c:0)^*.a]^{\omega}$

$p3 = [(x:0)^*.a]^{\omega}$

NEVER = {a,a}

- **Never access the bus twice simultaneously**

Example Synchronization



- **Synchronization between a sender and a receiver in a blocking protocol**
  - **Sender = (x : r)*.a**
  - **Receiver = (y : k)*.a**
  - **ALWAYS = {{a; a}}**
  - **NEVER = {{r; k}}**

Design with CFEs

- **Representation:**
  - A CFE can be compiled into a *specification automaton*
  - Representing all feasible behaviors
- **Synthesis:**
  - A control-unit implementation is a FSM
  - Derivable from a specification automaton by assigning values to decision variables over time
- **Optimization:**
  - Minimize a cost function defined over the decision variables

CFE Summary

- Control-flow expression are a modeling tool
- Formal semantic:
  - Support for synthesis and verification
- Synthesis path from CFEs to control-unit