SOFTWARE ARCHITECTURES AND ENGINEERING FOR EMBEDDED SYSTEMS

TOPIC 6

Josephine Magu MSC UON

ORGANIZATION

1. Model-based Development of Emb. Sys.
2. Review of models of concurrency in programming languages
3. Synchronous Model of concurrency
4. Introduction to Esterel
5. Advanced Features of Esterel
6. Simple case studies using Esterel
7. Verification
8. POLIS: A HW-SW co-design tool for ES

MODEL-BASED DEVELOPMENT OF EMBEDDED SYSTEMS

DEVELOPMENT CHALLENGES

Embedded Systems are quite complex

1. Correct functioning is crucial
   - safety-critical applications
   - damage to life, economy can result
2. They are Reactive Systems
   - Once started run forever.
   - Termination is a bad behavior.
   - Compare conventional computing

     (transformational systems)

DEVELOPMENT CHALLENGES

3. Concurrent systems
   - System and environment run concurrently
   - multi-functional

4. Real-time systems
   - not only rt. outputs but at rt. time
   - imagine a delay of few minutes in pacemaker system

Development Challenges

5. Stringent resource constraints
   - compact systems
     - simple processors
     - limited memory
   - quick response
   - good throughput
   - low power
   - Time-to-market

SYSTEM DEVELOPMENT

- Process of arriving at a final product from requirements
- Requirements
  - Vague ideas, algorithms, of-the shelf components, additional functionality etc.
  - Natural Language statements
  - Informal
- Final Products
  - System Components
  - Precise and Formal

SYSTEM COMPONENTS

- Embedded System Components
  - Programmable processors (controllers & DSP)
  - Standard and custom hardware
  - Concurrent Software
  - OS Components:
    - Schedulers, Timers, Watchdogs,
    - IPC primitives
  - Interface components
    - External, HW and SW interface

SYSTEM DEVELOPMENT

- Decomposition of functionality
- Architecture Selection: Choice of processors, standard hardware
- Mapping of functionality to HW and SW
- Development of Custom HW and software
- Communication protocol between HW and SW
- Prototyping, verification and validation

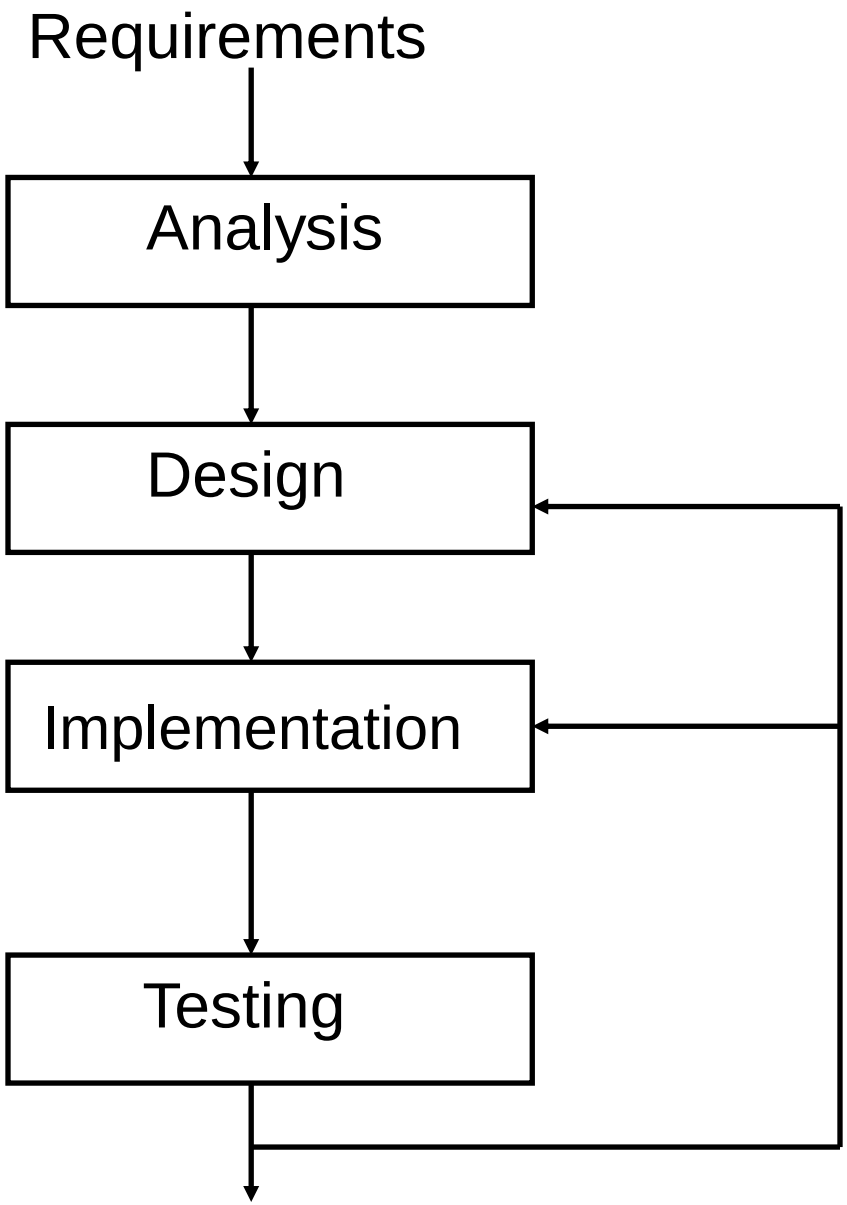
DESIGN CHOICES

- Choices in Components
  - Processors, DSP chips, Std. Components
- Many different choices in mapping
  - Fully HW solution
    - More speed, cost, TTM (Time to market), less robust
    - Std. HW development
  - Fully SW solution
    - Slow, less TTM, cost, more flexible
    - Std. Micro controller development

MIXED SOLUTION

- Desired Solution is often mixed
  - Optimal performance, cost and TTM
  - Design is more involved and takes more time
  - Involves Co-design of HW and SW
  - System Partitioning - difficult step
  - For optimal designs, design exploration and evaluation essential
  - Design practices supporting exploration and evaluation essential
  - Should support correctness analysis as it is crucial to ensure high quality

CLASSICAL DESIGN METHODOLOGY



DEVELOPMENT METHODOLOGY

- Simplified Picture of SW development
  - Requirements Analysis
  - Design
  - Implementation (coding)
  - Verification and Validation
  - Bugs lead redesign or re-implementation

DEVELOPMENT METHODOLOGY

- All steps (except implementation) are informal
  - Processes and objects not well defined and ambiguous
  - Design and requirement artifacts not precisely defined
  - Inconsistencies and incompleteness
  - No clear relationship between different stages
  - Subjective, no universal validity
  - Independent analysis difficult
  - Reuse not possible

CLASSICAL METHODOLOGY

- Totally inadequate for complex systems
  - Thorough reviews required for early bug removal
  - Bugs often revealed late while testing
  - Traceability to Design steps not possible
  - Debugging difficult
  - Heavy redesign cost
- Not recommended for high integrity systems
  - Read embedded systems

FORMAL METHODOLOGY

- A methodology using precisely defined artifacts at all stages
  - Precise statement of requirements
  - Formal design artifacts (Models)
  - Formal: Precisely defined syntax and semantics
  - Translation of Design models to implementation

MODEL-BASED DEVELOPMENT

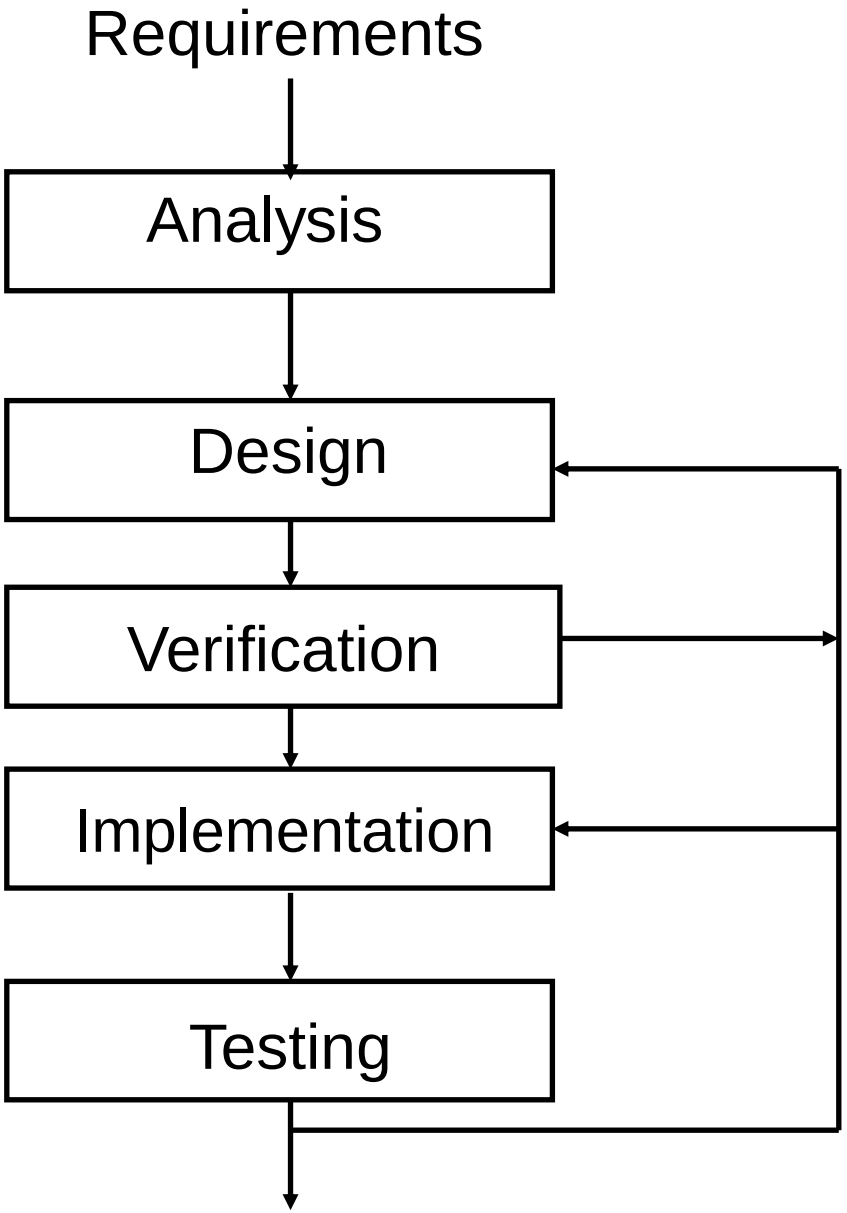
- Models are abstract and high level descriptions of design objects
- Focus on one aspect at a time
- Less development and redesign time
- Implementation constraints can be placed on models
- Design exploration, evaluation and quick prototyping possible using models

NEW PARADIGM

- Executable models essential
  - Simulation
- Can be rigorously validated
  - Formal Verification
- Models can be debugged and revised
- Automatic generation of final code
  - Traceability
- The paradigm

  Model – Verify – Debug – CodeGenerate

MODEL-BASED METHODOLOGY



TOOLS

- Various tools supporting such methodologies
- Commercial and academic
- POLIS (Berkeley), Cierto VCC (Cadence)
- SpecCharts (Irvine)
- STATEMATE, Rhapsody (ilogix)
- Rose RT (Rational)
- SCADE, Esterel Studio (Esterel Technologies)
- Stateflow and Simulink (Mathworks)

MODELING LANGUAGES

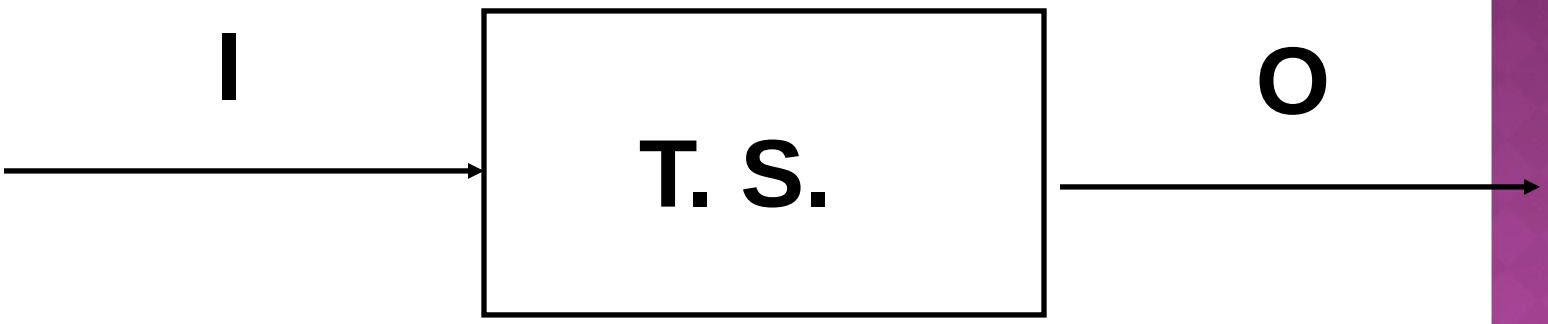
- Models need to be formal
- Languages for describing models
- Various languages exist
- High level programming languages (C, C++)
- Finite State Machines, Statecharts, SpecCharts, Esterel, Stateflow
- Data Flow Diagrams, Lustre, Signal, Simulink
- Hardware description languages (VHDL, Verilog)
- Unified Modeling Language(UML)

MODELING LANGUAGES

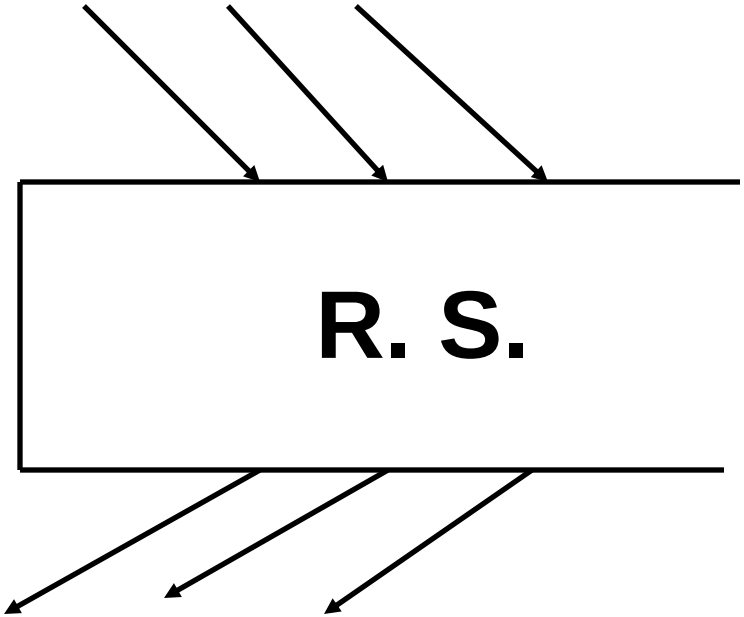
- Choice of languages depend upon the nature of computations modeled
- Seq. programming models for standard data processing computations
- Data flow diagrams for iterative data transformation
- State Machines for controllers
- HDLs for hardware components

REACTIVE SYSTEMS

- Standard Software is a transformational system
- Embedded software is reactive



REACTIVE SYSTEMS



REACTIVE SYSTEM FEATURES

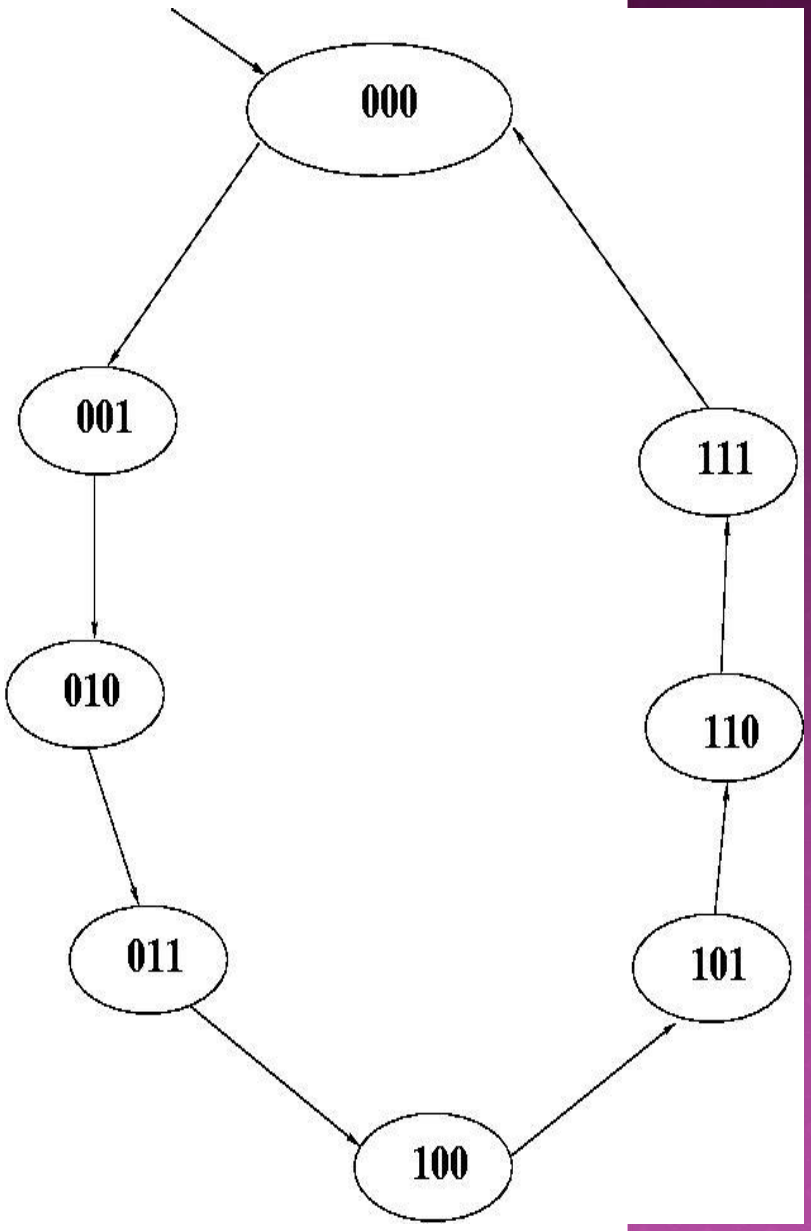
- Non-termination
- Ongoing continuous relationship with environment
- Concurrency (at least system and environment)
- Event driven
- Events at unpredictable times
- Environment is the master
- Timely response (hard and soft real time)
- Safety - Critical
- Conventional models inadequate

FINITE STATE MACHINES

- One of the well-known models
- Intuitive and easy to understand
- Pictorial appeal
- Can be made rigorous
- Standard models for Protocols, Controllers, HW

A SIMPLE EXAMPLE

- 3 bit counter
- C – count signal for increments
- Resets to 0 when counter reaches maximum value
- Counter can be described by a program with a counter variable (Software Model)
- Or in detail using flip flops, gates and wires



STATE MACHINE MODEL

- Counter behaviour naturally described by a state machine
- States determine the current value of the counter
- Transitions model state changes to the event C.
- Initial state determines the initial value of the counter
- No final state (why?)

PRECISE DEFINITION

< Q, q0, S, T>

- Q – A finite no. of state names
- q0 – Initial state
- S – Edge alphabet

Abstract labels to concrete event, condition and action

- T – edge function or relation

SEMANTICS

- Given the syntax, a precise semantics can be defined
- Set of all possible sequences of states and edges
- Each sequence starts with the initial state
- Every state-edge-state triples are adjacent states connected by an edge
- Given a FSM, a unique set of sequences can be associated
- Language accepted by a FSM

ABSTRACT MODELS

- Finite State machine model is abstract
- Abstracts out various details
  - How to read inputs?
  - How often to look for inputs?
  - How to represent states and transitions?
  - Focus on specific aspects
- Easy for analysis, debugging
- Redesign cost is reduced
- Different possible implementations
  - Hardware or Software
  - Useful for codesign of systems

INTUITIVE MODELS

- FSM models are intuitive
- Visual
  - A picture is worth a thousand words
- Fewer primitives – easy to learn, less scope for mistakes and confusion
- Neutral and hence universal applicability
  - For Software, hardware and control engineers

RIGOROUS MODELS

- FSM models are precise and unambiguous
- Have rigorous semantics
- Can be executed (or simulated)
- Execution mechanism is simple: An iterative scheme

```
state = initial_state
loop
    case state:
        state 1:   Action 1
        state 2:   Action 2
            . . .
    end case
end
```

CODE GENERATION

- FSM models can be refined to different implementation
  - Both HW and SW implementation
  - Exploring alternate implementations
  - For performance and other considerations
- Automatic code generation
- Preferable over hand generated code
- Quality is high and uniform

STATES AND TRANSITIONS

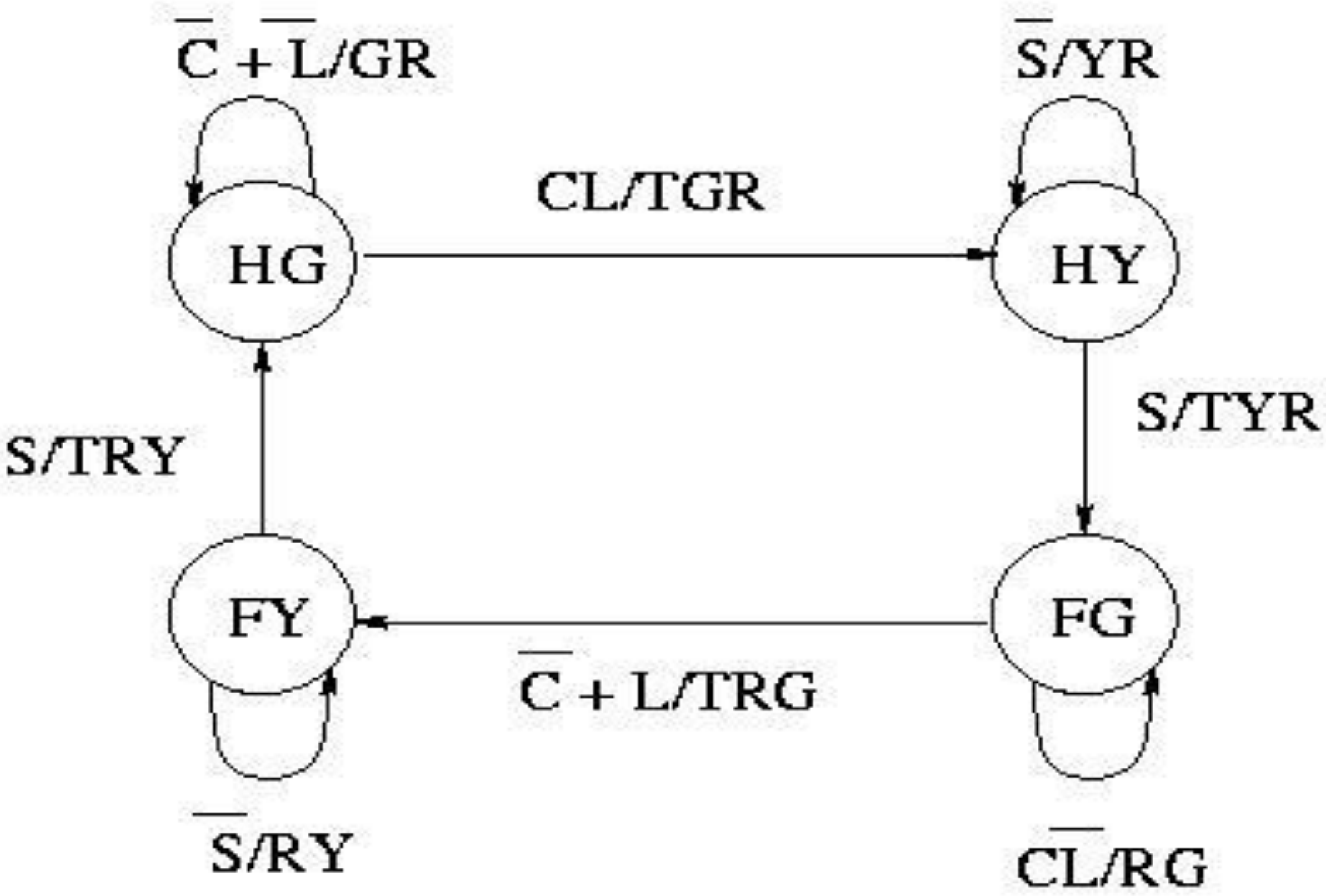
- Many Flavors of State Machines
  - edge labeled - Mealy machines
  - state labeled - Kripke structures
  - state and edge labeled - Moore machines
  - Labels
    - Boolean combination of input signals and outputs
    - communication events (CSP, Promela)

ANOTHER EXAMPLE

A Traffic Light Controller

- Traffic light at the intersection of High Way and Farm Road
- Farm Road Sensors (signal C)
- G, R – setting signals green and red
- S,L - Short and long timer signal
- TGR - reset timer, set highway green and farm road red

STATE MACHINE



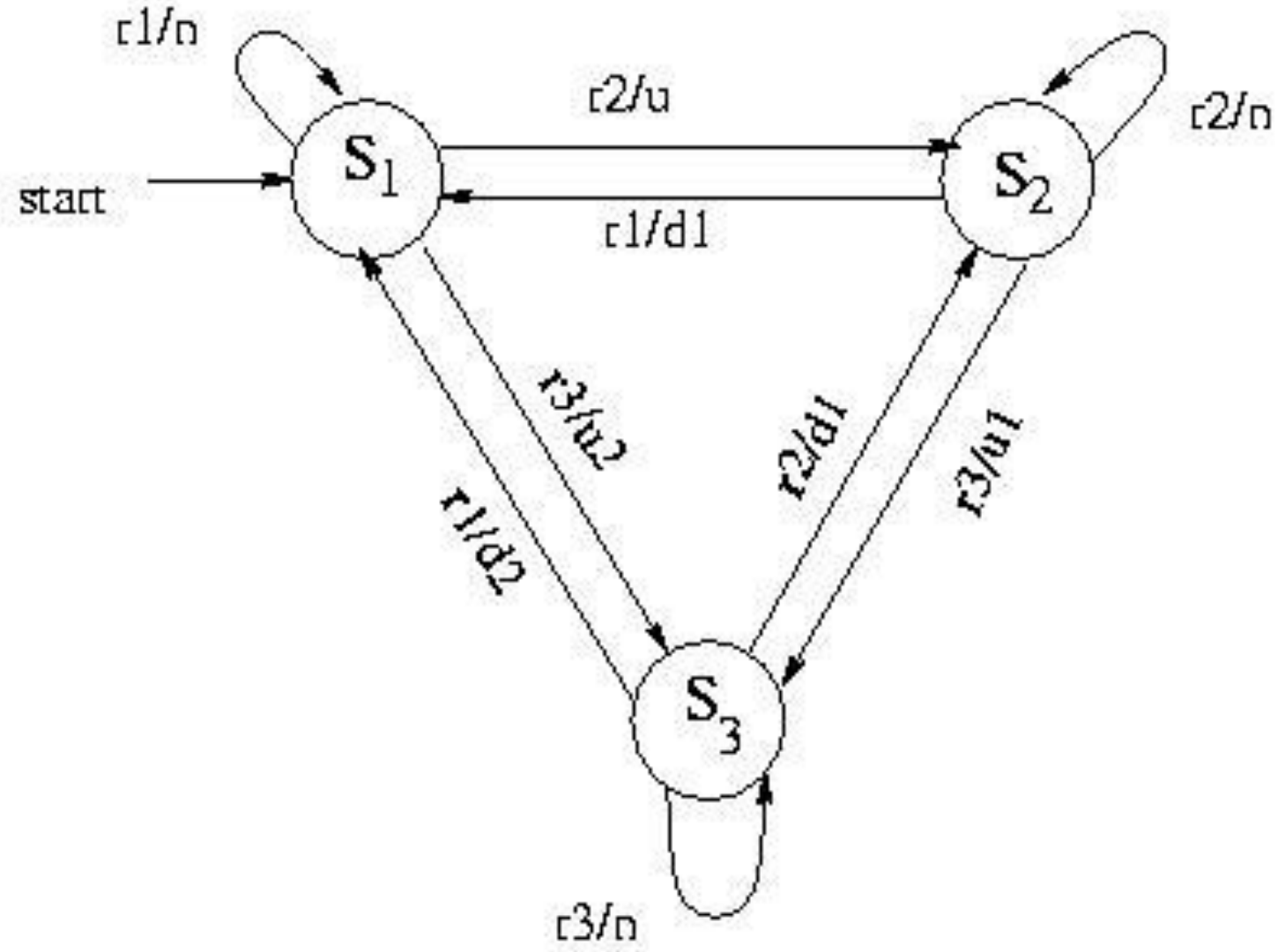
ANOTHER EXAMPLE

A Simple Lift Controller

3-floor lift

- Lift can be in any floor
  - S_i - in floor I
- Request can come from any floor
  - ri - request from floor I
- Lift can be asked to move up or down
  - uj,dj - up/down to jth floor

FSM MODEL



NONDETERMINISM

- Suppose lift is in floor 2 (State S_2)
- What is the next state when when requests r1 and r3 arrive?
  - Go to S_1
  - Or go to S_3
- The model non committal – allows both
- More than one next state for a state and an input
- This is called nondeterminism
- Nondeterminism arises out of abstraction
- Algorithm to decide the floor is not modeled

NONDETERMINISM

- Models focus attention on a particular aspect
- The lift model focussed on safety aspects
- And so ignored the decision algorithm
  - Modeling languages should be expressive
  - Std. Programming languages are not
- Use another model for capturing decision algorithm
- Multiple models, separation of concerns
  - Independent analysis and debugging
  - Management of complexity
- Of course, there should be a way of

C-MODEL

```
  enum floors {f1, f2, f3};
enum State {first, second, third};
enum bool {ff, tt};
enum floors req, dest;
enum bool up, down = ff;
enum State cur_floor = first;

req = read_req();

  while (1)
{ switch (cur_floor)
  { case first: if (req == f2)
                   {up = tt; dest = f2;}
                else if (req == f3)
                   {up = tt; dest = f3;}
                else { up == ff; down = ff;};
                break;
```

C- MODEL

```
case second: if (req == f3)
                    {up = tt; dest = f3;}
                  else if (req == f1)
                          { up = ff; down = tt; dest
  = f1;}
                          else { up == ff; down =
  ff;};
                  break;
case third:   if (req == f2)
                   {up = ff; down = tt; dest = f2;}
                 else if (req == f1)
                          { up = ff; down = tt; dest =
  f1;}
                          else { up == ff; down = ff;};
```

SUITABILITY OF C

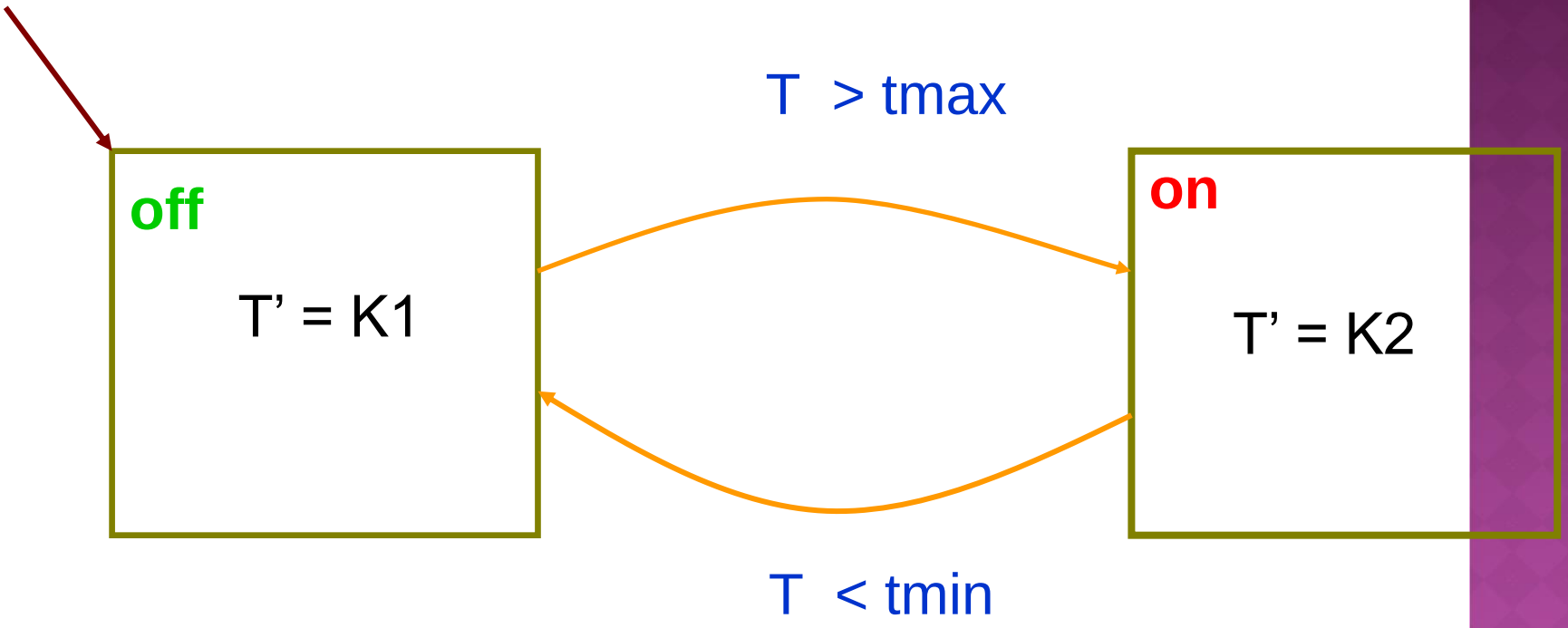
- C not natural for such applications
- Various problems
  - Events and states all modeled as variables
  - Not natural for even oriented embedded applications
  - States are implicit (control points decide the states)
  - No abstract description possible
  - Commitment to details at an early stage
  - Too much of work when the design is likely to be discarded

EXERCISE

- Is the C model non-deterministic?
- What happens when two requests to go in different directions arrive at a state?

YET ANOTHER EXAMPLE

- A Simple Thermostat controller



SUMMARY

- Finite number of states
- Initial state
- No final state (reactive system)
- Non determinism (result of abstraction)
- Edges labeled with events
- Behavior defined by sequences of transitions
- Rigorous semantics
- Easy to simulate and debug
- Automatic Code generation

PROBLEMS WITH FSMS

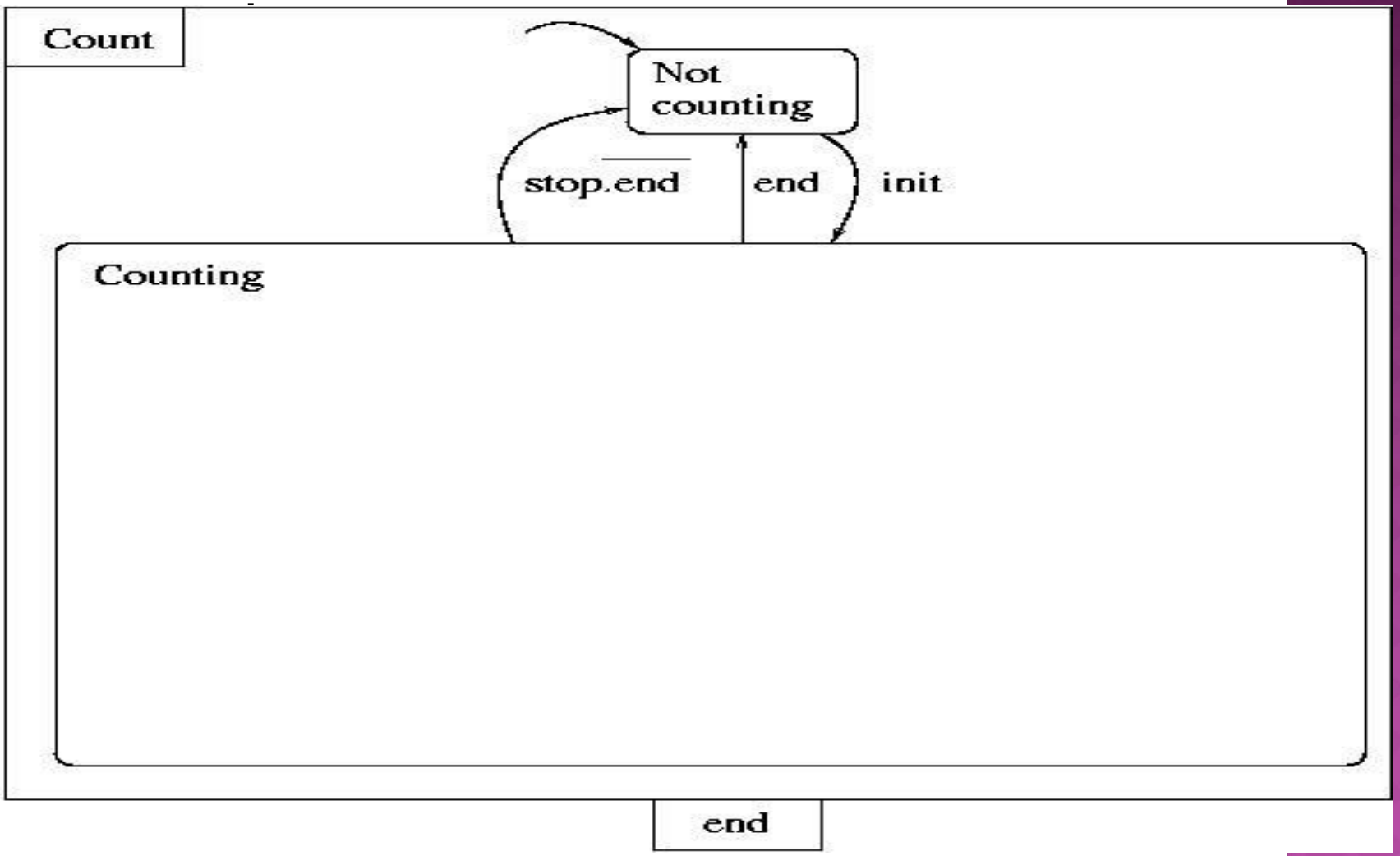
- All is not well with FSMs
- FSMs fine for small systems (10s of states)
- Imagine FSM with 100s and 1000s of states which is a reality
- Such large descriptions difficult to understand
- FSMs are flat and no structure
- Inflexible to add additional functionalities
- Need for structuring and combining different state machines

STATECHARTS

- Extension of FSMs to have these features
- Due to David Harel
- Retains the nice features
  - Pictorial appeal
  - States and transitions
- enriched with two features
  - Hierarchy and Concurrency
- States are of two kinds
  - OR state (Hierarchy)
  - AND state (concurrency)

OR STATES

- An OR state can have a whole state machine inside it



OR STATES

- When the system is in the state Count, it is either in counting or not_counting
- exactly in one of the inner states
- Hence the term OR states (more precisely XOR state)
- When Count is entered, it will enter not_counting
  - default state
- Inner states can be OR states (or AND states)

OR STATES

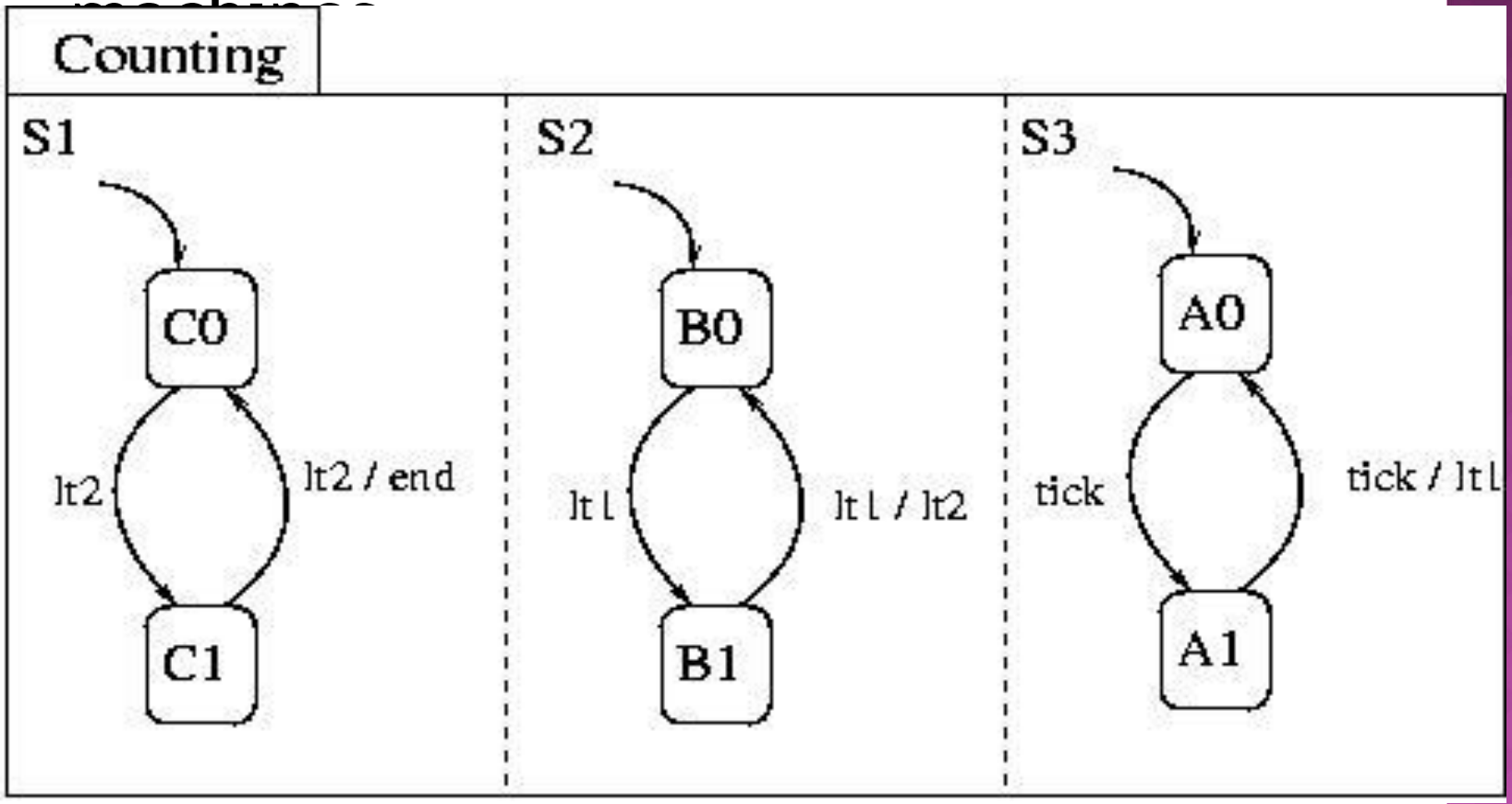
- Both outer and inner states active simultaneously
- When the outer state exits, inner states also exited
- Priorities of transitions
- Preemption (strong and weak)

ECONOMY OF EDGES

- Every transition from outer state corresponds to many transitions from each of the inner states
- Hierarchical construct replaces all these into one single transition
- Edge labels can be complex

AND STATES

- An Or state contains exactly one state machine
- An And state contains two or more state machines



EXAMPLE

- Counting is an And state with three state machines
- S1, S2, S3, concurrent components of the state
- When in state Counting, control resides simultaneously in all the three state machines
- Initially, control is in C0, B0 and A0
- Execution involves, in general, simultaneous transitions in all the state machines

EXAMPLE (CONTD.)

- When in state C0, B1, A2, clock signal triggers the transition to B2 and A2 in S2 and S3
- When in C0, B2, A2, clock signal input trigger the transitions to C1, B0 and A0 in all S1, S2, S3
- And state captures concurrency
- Default states in each concurrent component

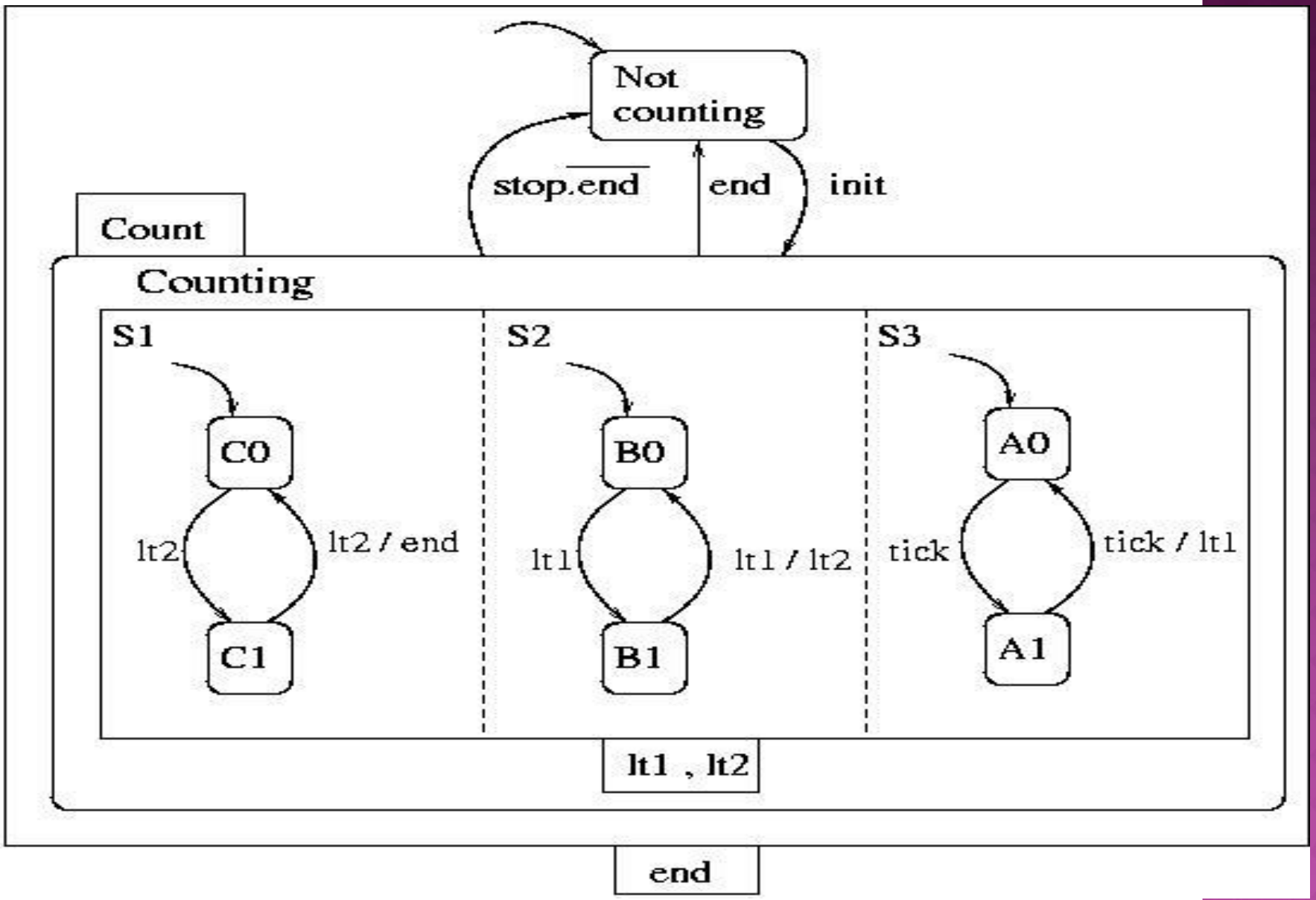
ECONOMY OF STATES

- An AND-state can be flattened to a single state machine
- Will result in exponential number of states and transitions
- AND state is a compact and intuitive representation

COUNTING

- What are the three components of the state?
- They represent the behaviour of the three bits of a counter
- S3 – the least significant bit, S2 the middle one and S1 the most significant bit
- Compare this with the flat and monolithic description of counter state machine given earlier
- Which is preferable?
- The present one is robust - can be redesigned to accommodate additional

COMPLETE MACHINE



COMMUNICATION

- Concurrent components of AND state communicate with each other
- Taking an edge requires certain events to occur
- New signals are generated when an edge is taken
- These can trigger further transitions in other components
- A series of transitions can be taken as a result of one transition triggered by environment event
- Different kinds of communication primitives

FLAT STATE MACHINES

- Capture the behaviour of the counter using FSMs
- Huge number of states and transitions
- Explosion of states and transitions
- Statechart description is compact
- Easy to understand
- Robust
- Can be simulated
- Code generation is possible
- Execution mechanism is more complex

EXERCISE

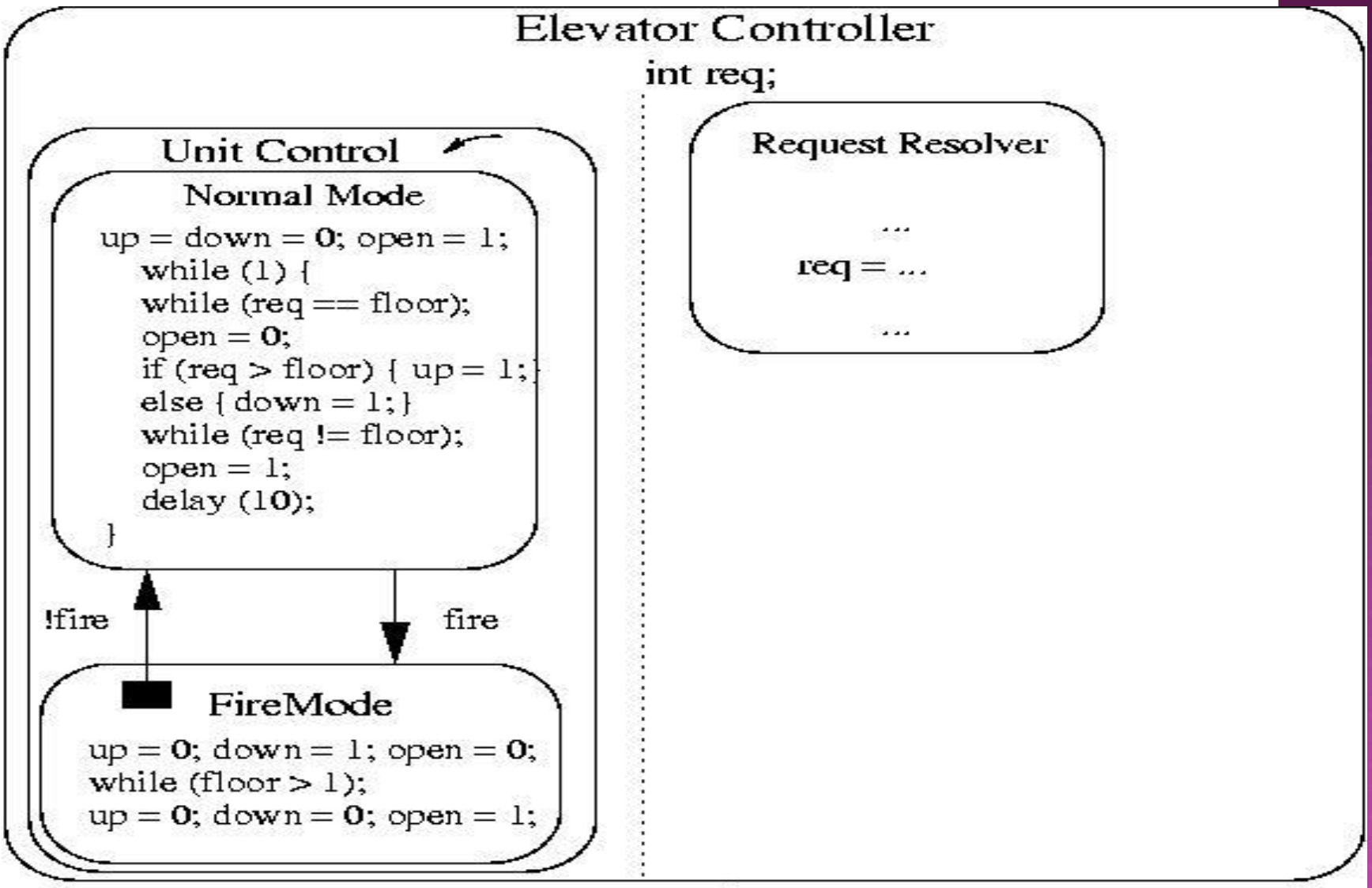
- Extend the lift controller example
  - Control for closing and opening the door
  - Control for indicator lamp
  - Avoid movement of the lift when the door is open
  - Include states to indicate whether the lift is in service or not
  - Controller for multiple lifts
- Give a statechart description

EXTENSIONS TO STATECHARTS

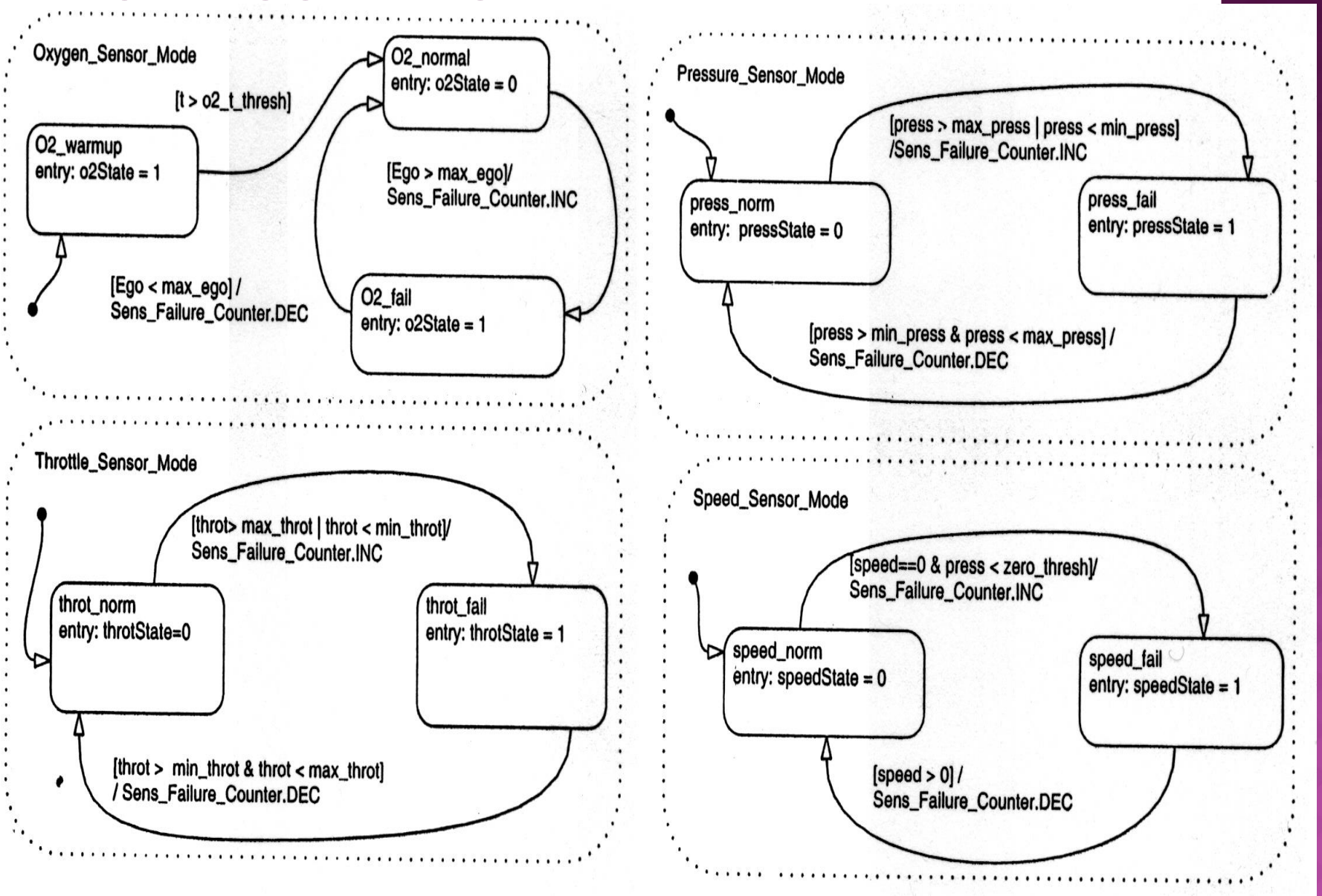
- various possibilities explored
- adding code to transitions
- to states
- complex data types and function calls
- Combining textual programs with statecharts
- Various commercial tools exist
- Statemate and Rhapsody (ilogix)
- UML tools (Rational rose)
- Stateflow (Mathworks)
- SynchCharts (Esterel Technologies)

EXAMPLE

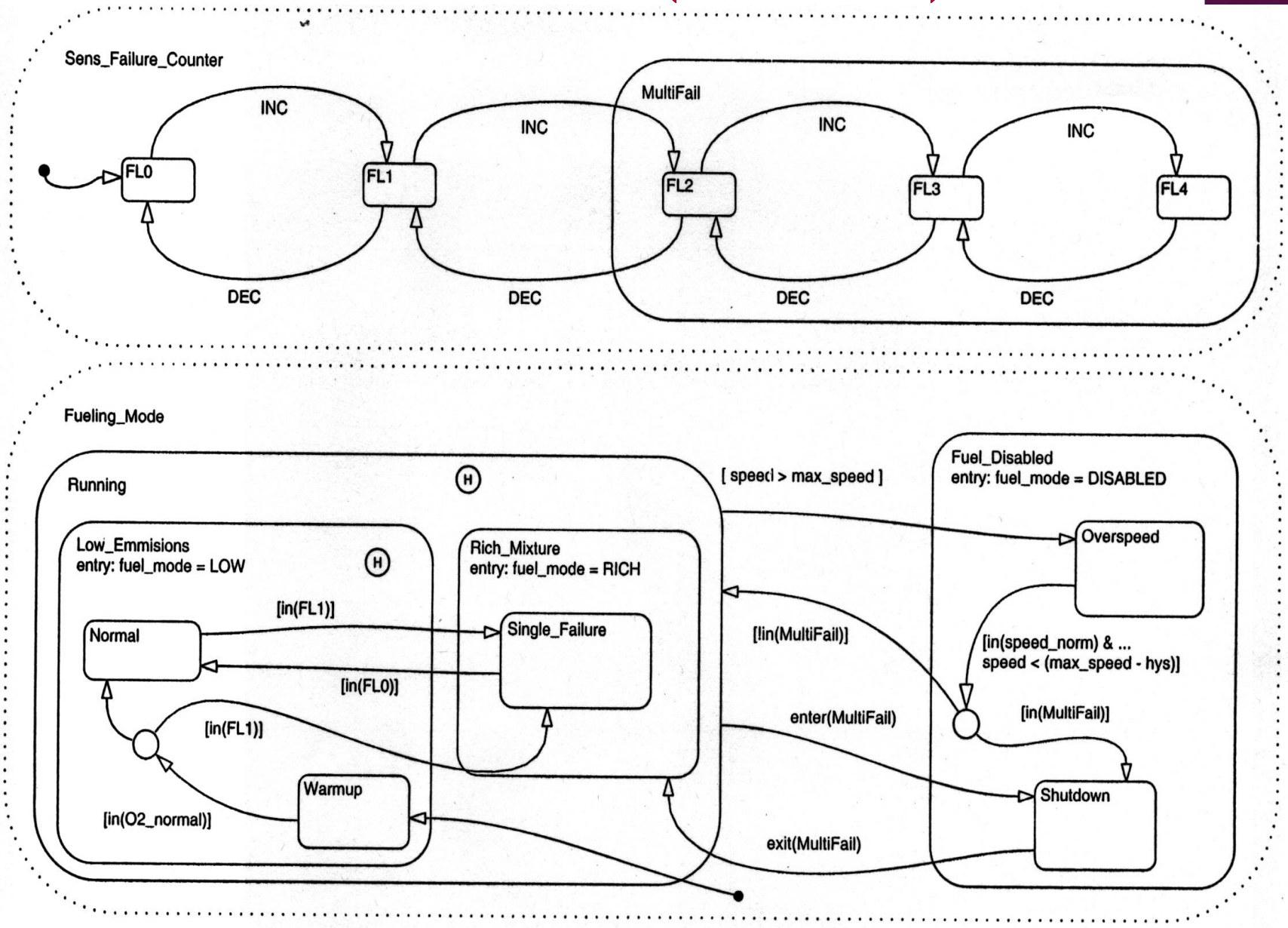
- Program State Machine model



FUEL CONTROLLER



FUEL CONTROLLER (CONTD.)



MORE EXERCISES

- Construct the State machine models of
  - Critical Section Problem
  - Producer-Consumer Problem
  - Dining Philosopher Problem
- And argue the correctness of solutions
- Formal Analysis and Verification (more on this later)

OTHER MODELS

- Synchronous Reactive Models
  - useful for expressing control dominated application
  - rich primitives for expressing complex controls
  - Esterel (Esterel Technologies)
  - More on this later

DESIGN FEATURES

- Two broad classifications
  - Control-dominated designs
  - Data-dominated Designs

- Control-dominated designs
  - Input events arrive at irregular and unpredictable times
  - Time of arrival and response more crucial than values

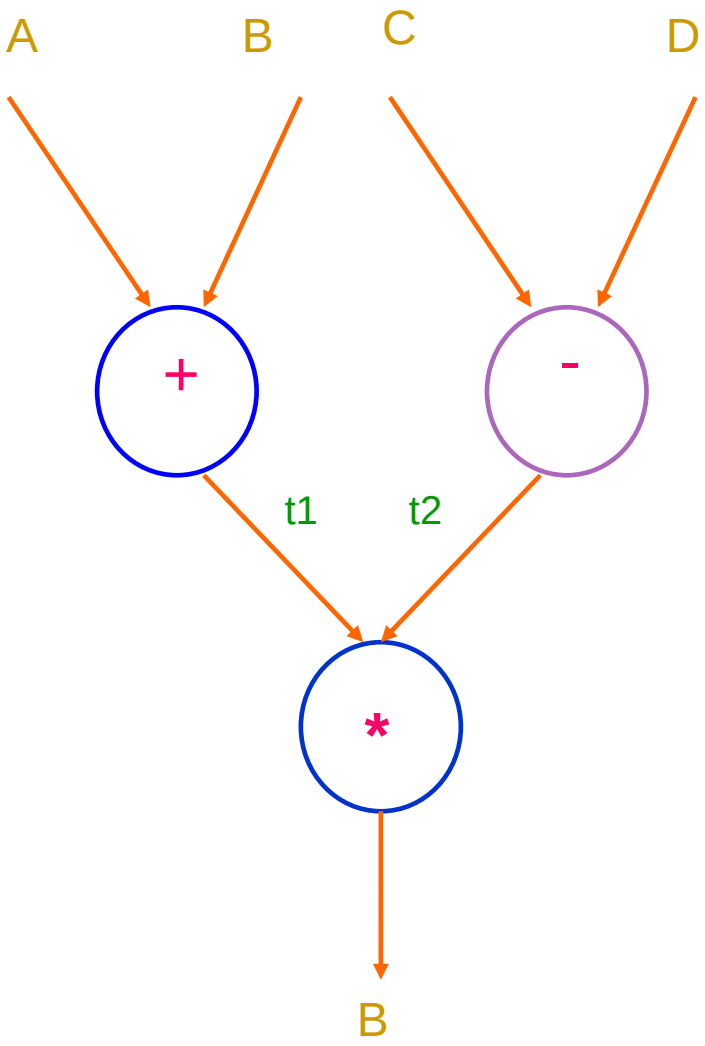
DESIGN FEATURES

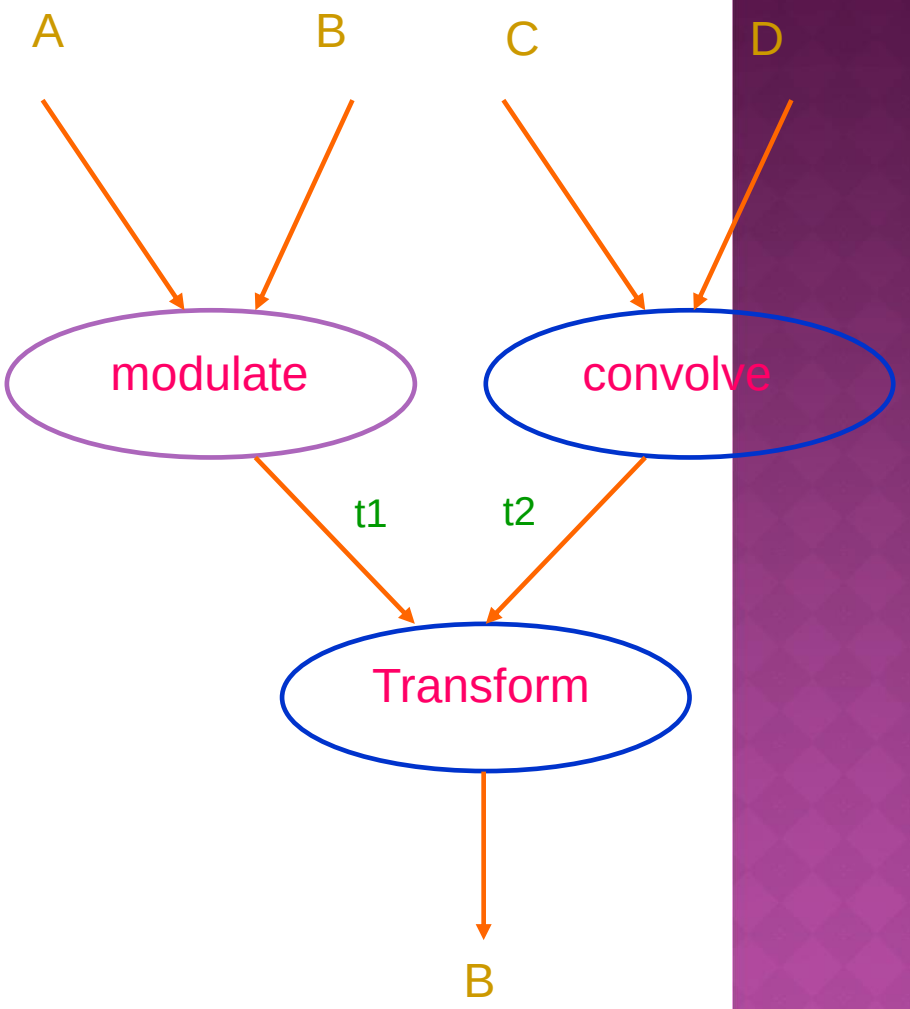
- Data-dominated designs
  - Inputs are streams of data coming at regular intervals (sampled data)
  - Values are more crucial
  - Outputs are complex mathematical functions of inputs
  - numerical computations and digital signal processing computations

DATA FLOW MODELS

- State machines, Statecharts, Esterel are good for control-dominated designs
- Date flow models are useful for data-dominated systems
- Special case of concurrent process models
- System behaviour described as an interconnection of nodes
- Each node describes transformation of data
- Connection between a pair of nodes

EXAMPLE





DATA FLOW MODELS

- Graphical Languages with support for
  - Simulation, debugging, analyisis
  - Code generation onto DSP and micro processors
- Analysis support for hardware-software partitioning
- Many commercial tools and languages
  - Lustre, Signal
  - SCADE

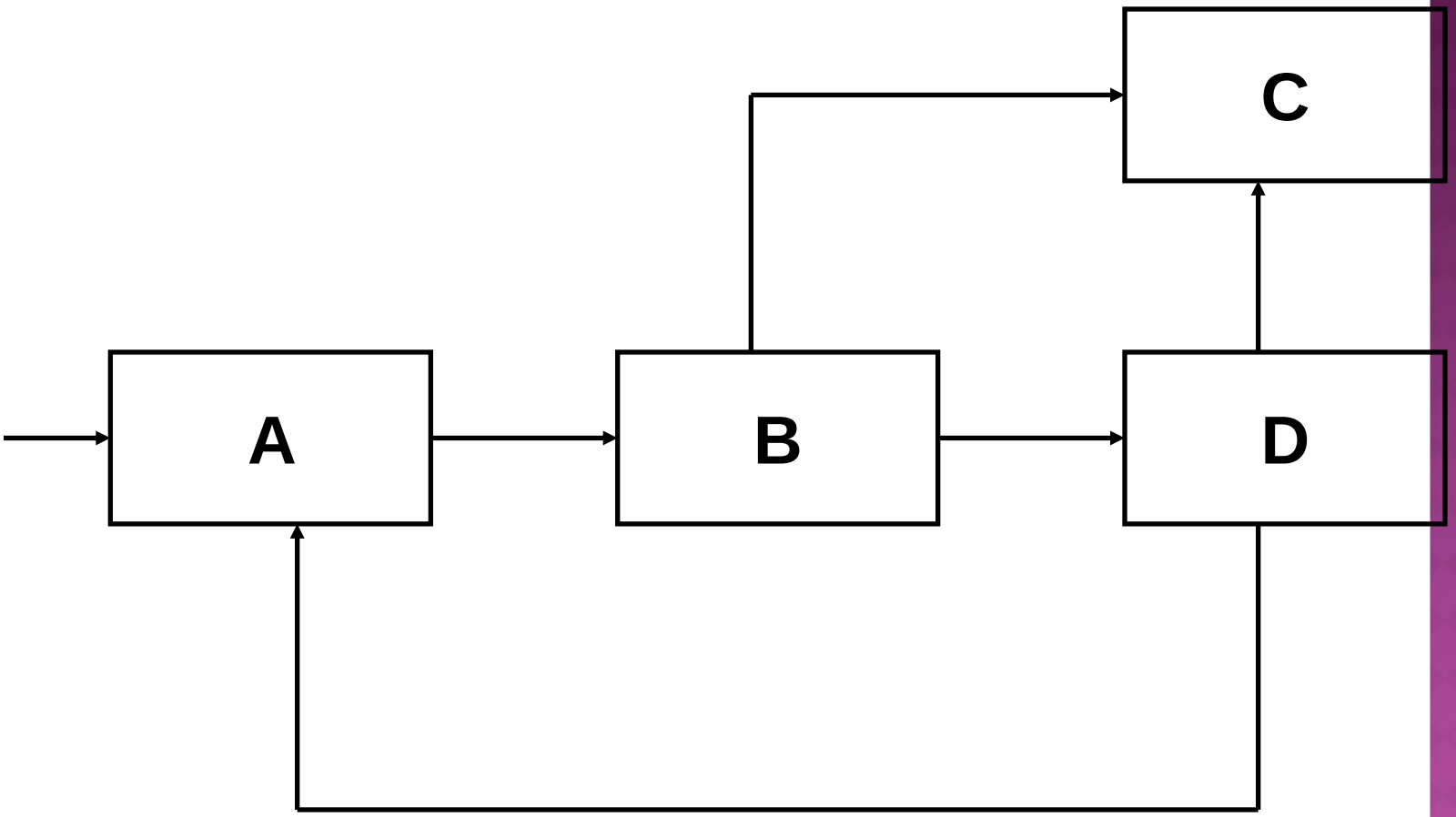
DISCRETE EVENT MODELS

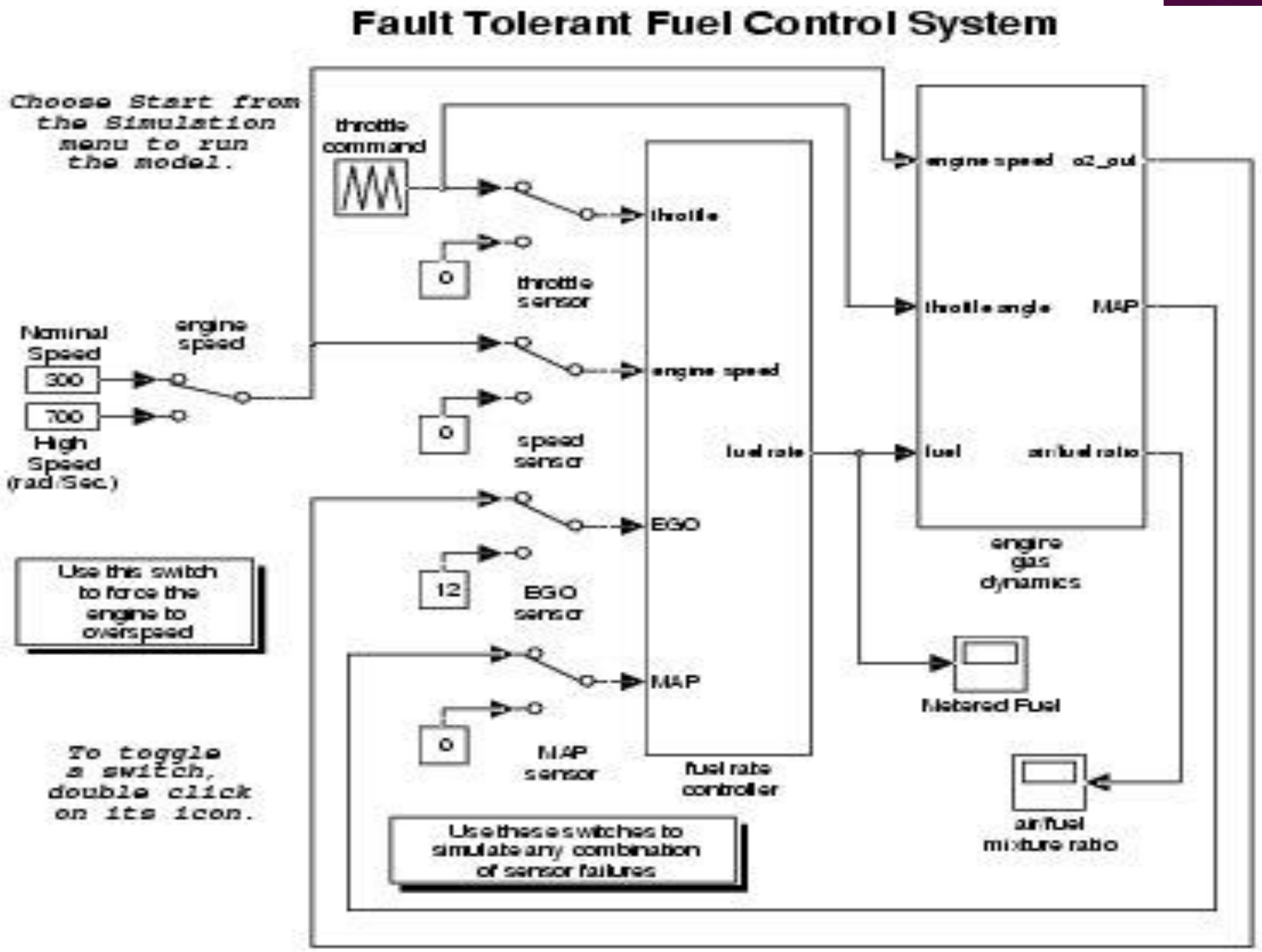
- Used for HW systems
- VHDL, Verilog
- Models are interconnection of nodes
- Each node reacts to events at their inputs
- Generates output events which trigger other nodes

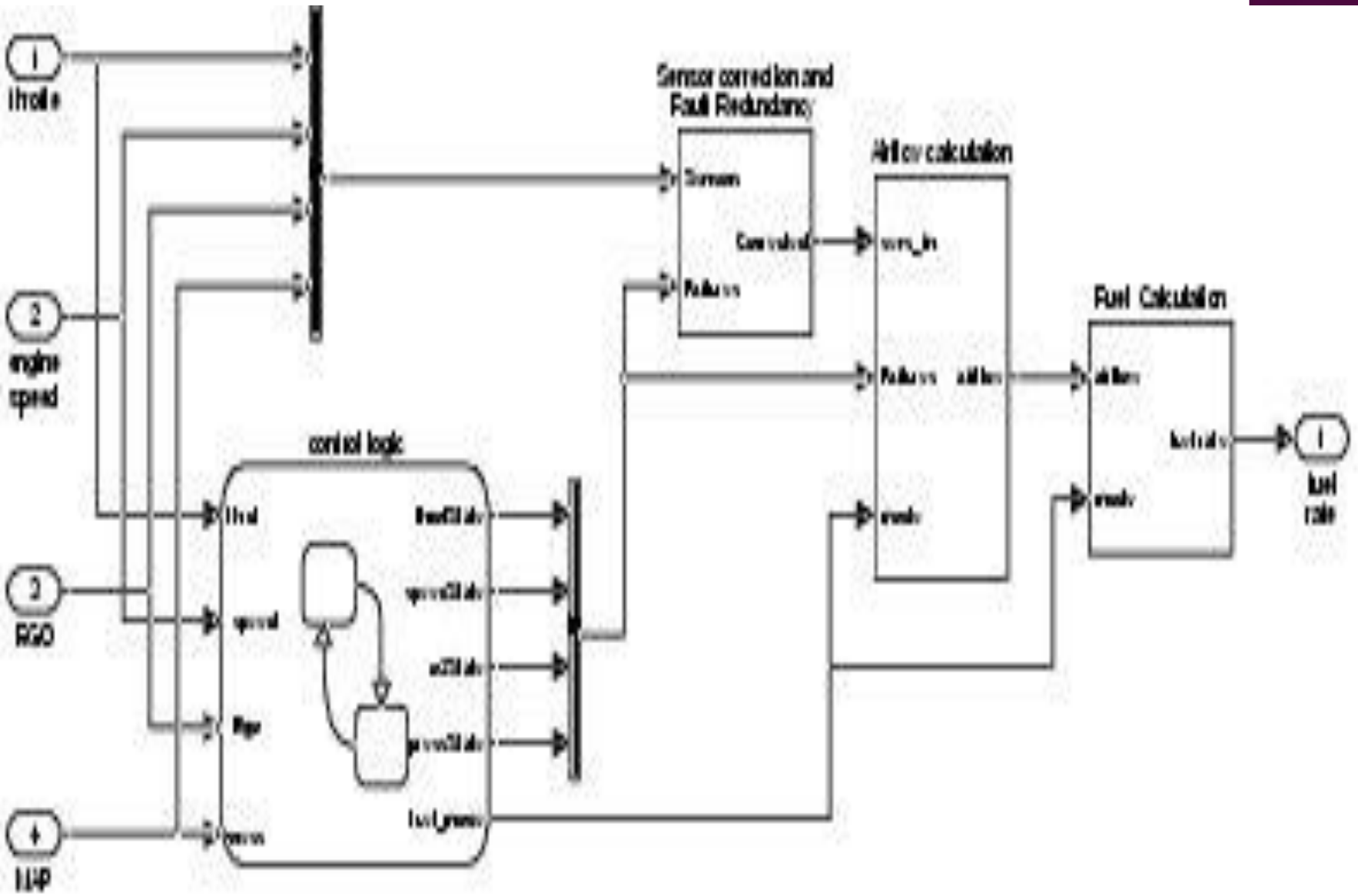
DE MODELS

- External events initiates a reaction
- Delays in nodes modeled as delays in event generation
- Simulation
- Problems with cycles
- Delta cycles in VHDL

DISCRETE EVENT MODELS




Fault Tolerant Fuel Control System
Choose Start from the Simulation menu to run the model.
throttle command
throttle
0
throttle sensor
Nominal Speed
300
700
High Speed (rad/Sec)
engine speed
engine speed
0
speed sensor
fuel rate
EGO
12
EGO sensor
Use this switch to force the engine to overspeed
MAP
0
MAP sensor
fuel rate controller
Use these switches to simulate any combination of sensor failures
To toggle a switch, double click on its icon.
engine speed
o2_out
throttle angle
MAP
fuel
air/fuel ratio
engine gas dynamics
Metered Fuel
air/fuel mixture ratio



fuel rate controller

SOME MORE EXERCISE

- Give a more detailed model of the digital camera
  - Only certain data flow aspect of the camera is given in the class (and in the book)

SUMMARY

- Various models reviewed
  - Sequential programming models
  - Hierarchical and Concurrent State Machines
  - Data Flow Models, Discrete Event Models
- Each model suitable for particular applications
- State Machines for event-oriented control systems

SUMMARY

- Sequential program model, data flow model for function computation
- Real systems often require mixture of models
- Modeling tools and languages should have combination of all the features
  - Ptolemy (Berkeley)

REFERENCES

- F. Balarin et al., Hardware – Software Co-design of Embedded Systems: The POLIS approach, Kluwer, 1997
- N. Halbwachs, Synch. Prog. Of Reactive Systems, Kluwer, 1993
- D. Harel et al., STATEMATE: a working environment for the development of complex reactive systems, IEEE Trans. Software Engineering, Vol. 16 (4), 1990.
- J. Buck, et al., Ptolemy: A framework for simulating and prototyping heterogeneous systems, Int. Journal of Software Simulation, Jan. 1990