

# CS684

## Embedded Systems (Software)

### Models & Tools for Embedded Systems (I)

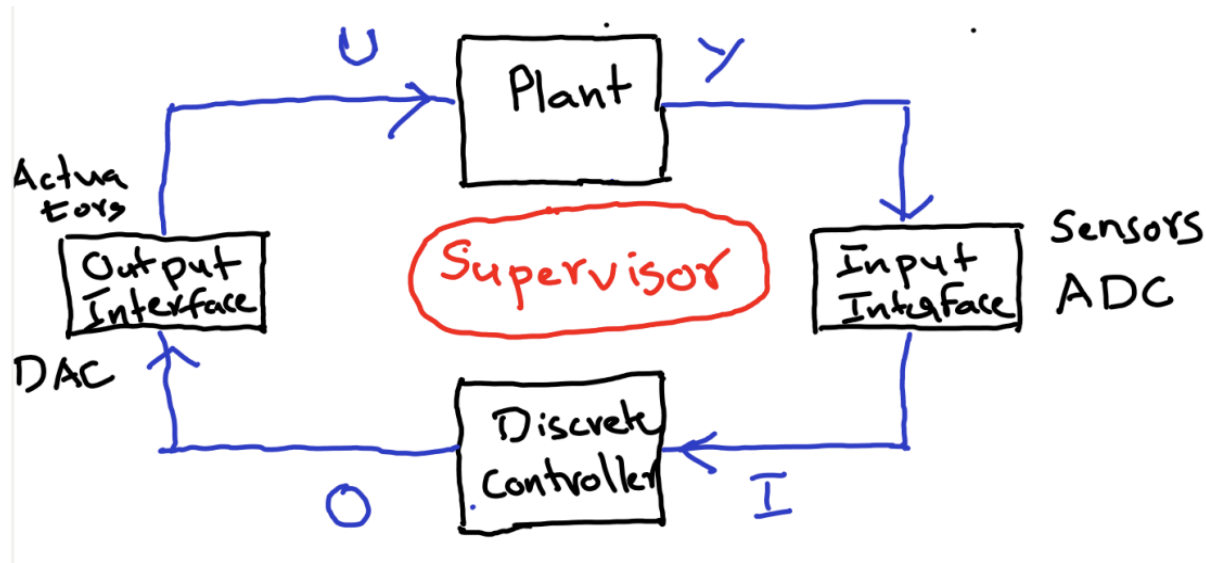
*Kavi Arya*  
CSE/ IIT Bombay



# Models and Tools for Embedded Systems

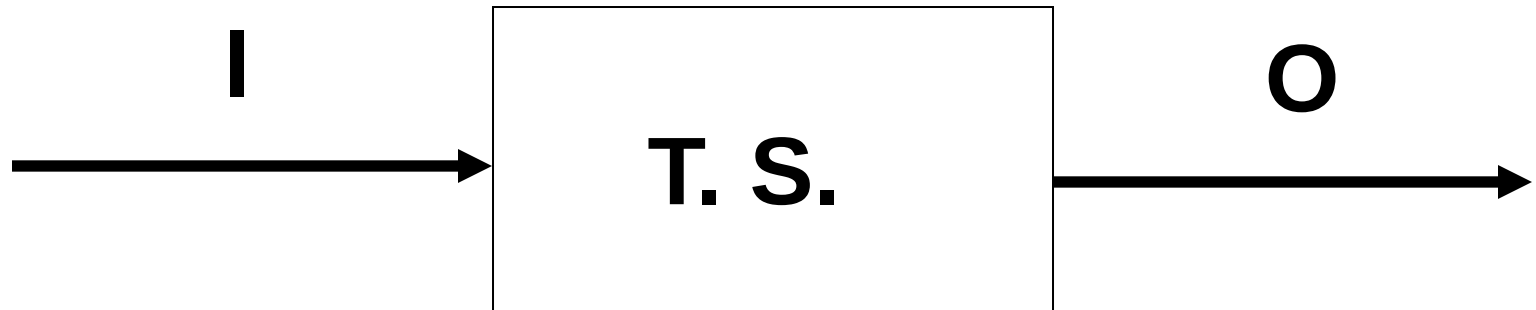
# Organization

1. Model-based Development of Embedded Sys.
2. Review of models of concurrency in programming languages
3. Introduction to Lustre/Heptagon
4. Simple case studies

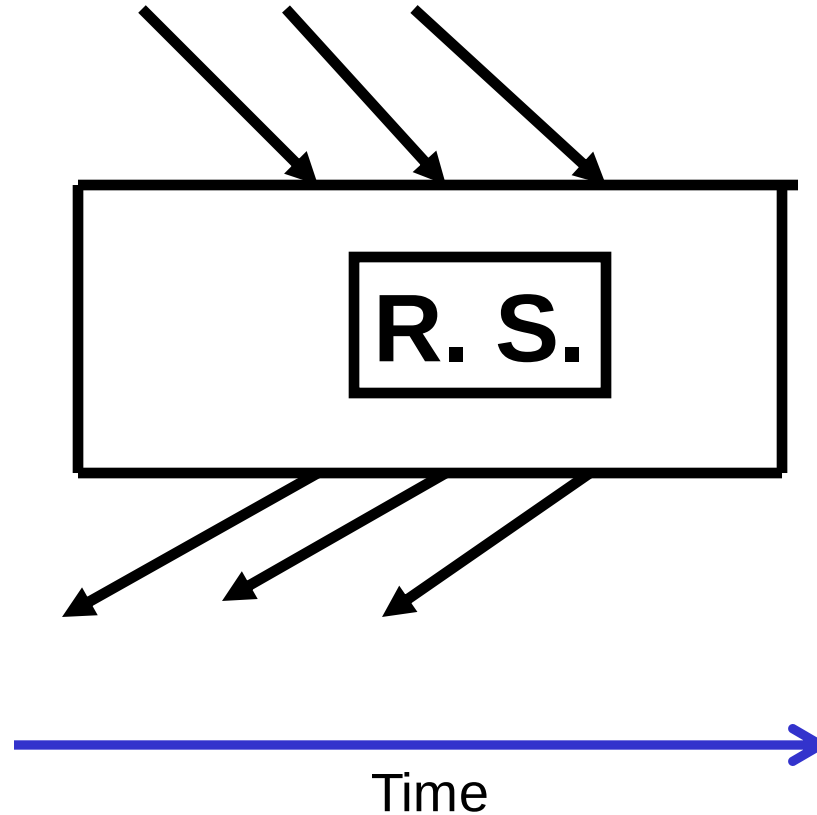


# Reactive Systems

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



# Reactive Systems



# Reactive Systems (features)

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

# Cyber Physical Systems

- Consists of physical elements controlled by
  - Supervisor, Discrete Controller & I/O interfaces
- Supervisor uses I/O Interface Drivers to interact w/devices,
- Control uses repeated execution of **Sense+Compute** Step
- Discrete Controller transforms input flow into output flow (synchronously)
- Discrete controller given as “**reactive kernel**” (“difficult” bit)
- Let’s explore how to describe this reactive kernel

# Development Challenges (Complexity)

- **Correct functioning is crucial**
- **Reactive**
- **Concurrent**
- **Realtime**
- **Stringent resource constraints**

# Development Challenges

Embedded Systems are 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 realtime outputs - but in realtime
- Imagine delay of 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 final product from reqs
- Requirements
  - Vague ideas, algorithms, of-the shelf components, additional functionality etc.
  - Natural Language statements
  - Informal
- Final Products
  - System Components
  - Precise and Formal

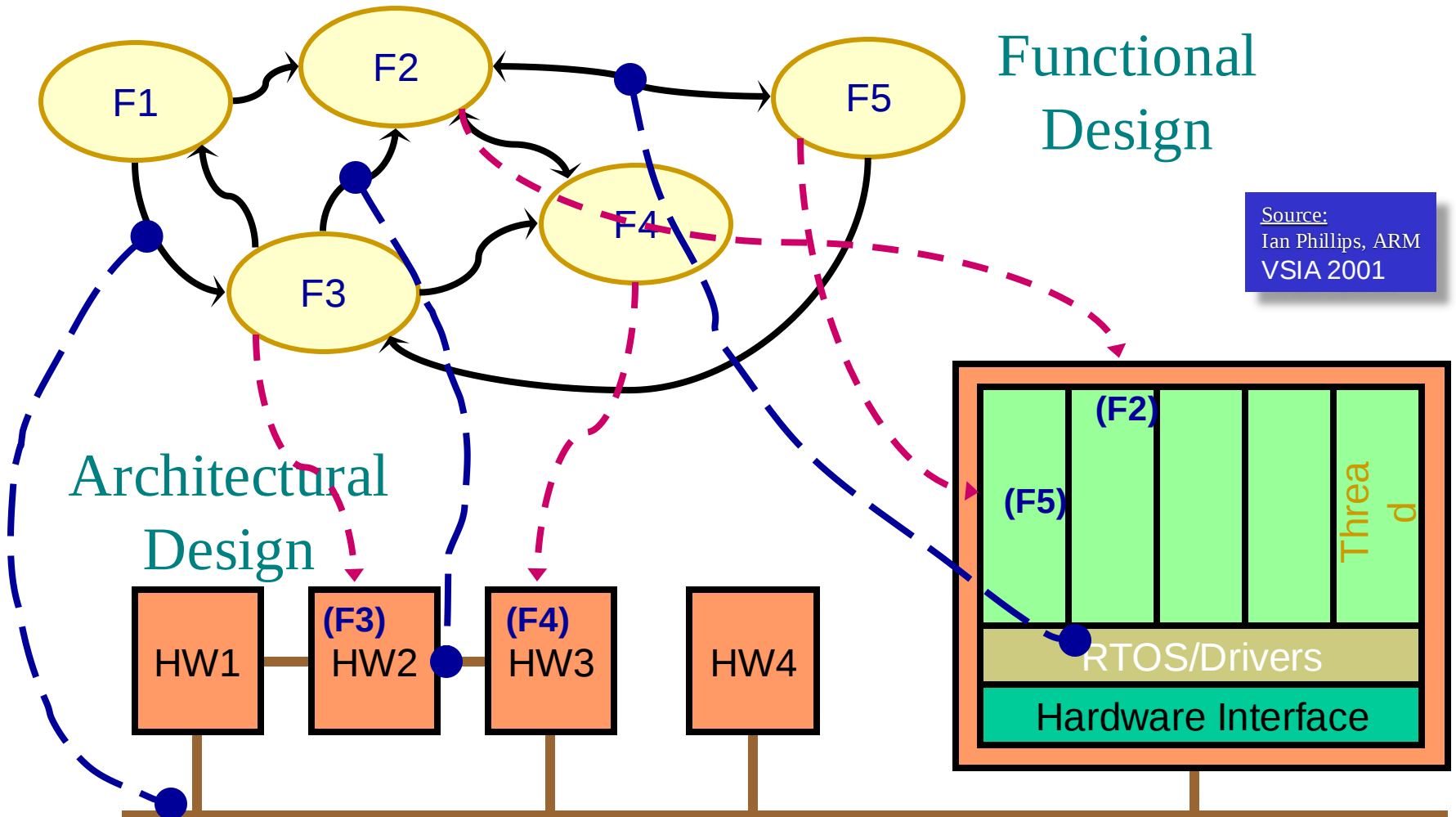
# System Components

- **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

# Functional Design & Mapping



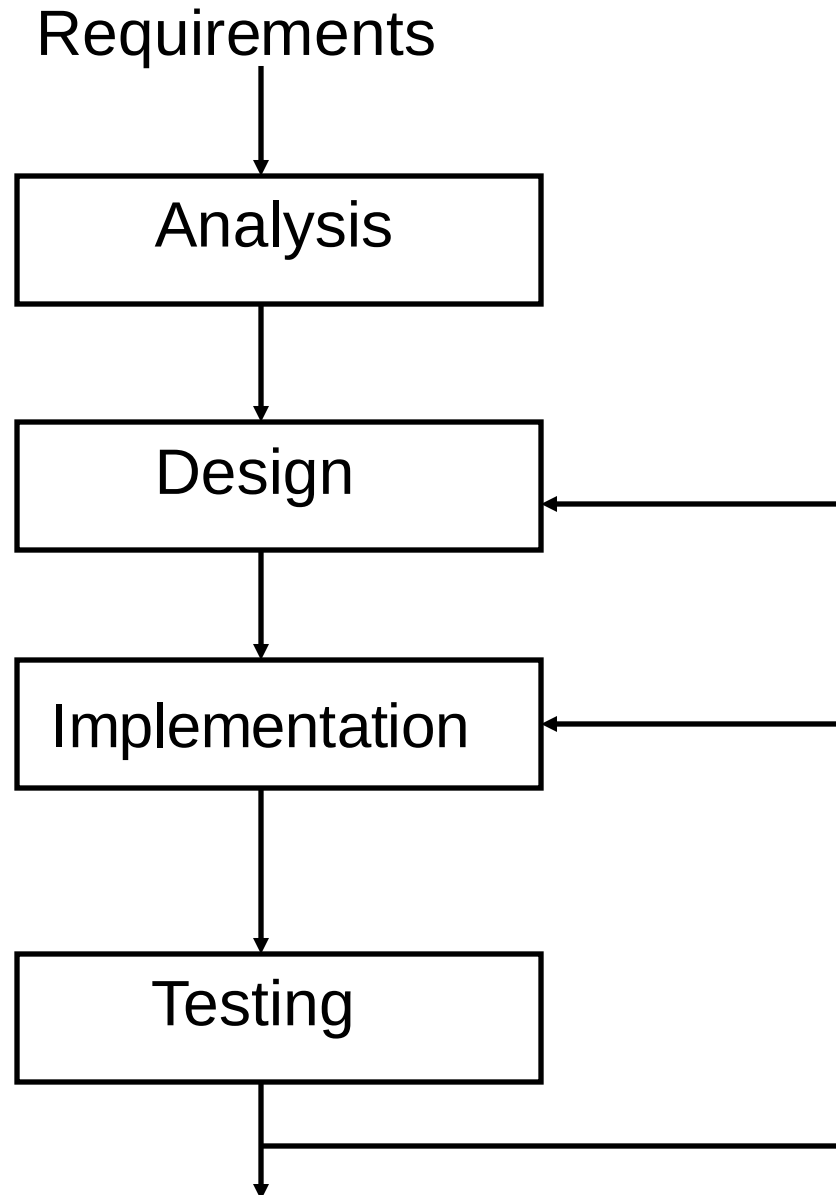
# Design Choices

- Choices in Components
  - Processors, DSP chips, standard components
- Many different choices in mapping
  - Fully HW solution
    - More speed, higher cost, longer TTM (“Time To Market”), less robust
    - Standard HW development
  - Fully SW solution
    - Slow, less TTM, less cost, more flexible
    - Standard microcontroller 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 & 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/ 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**
  - i.e. 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

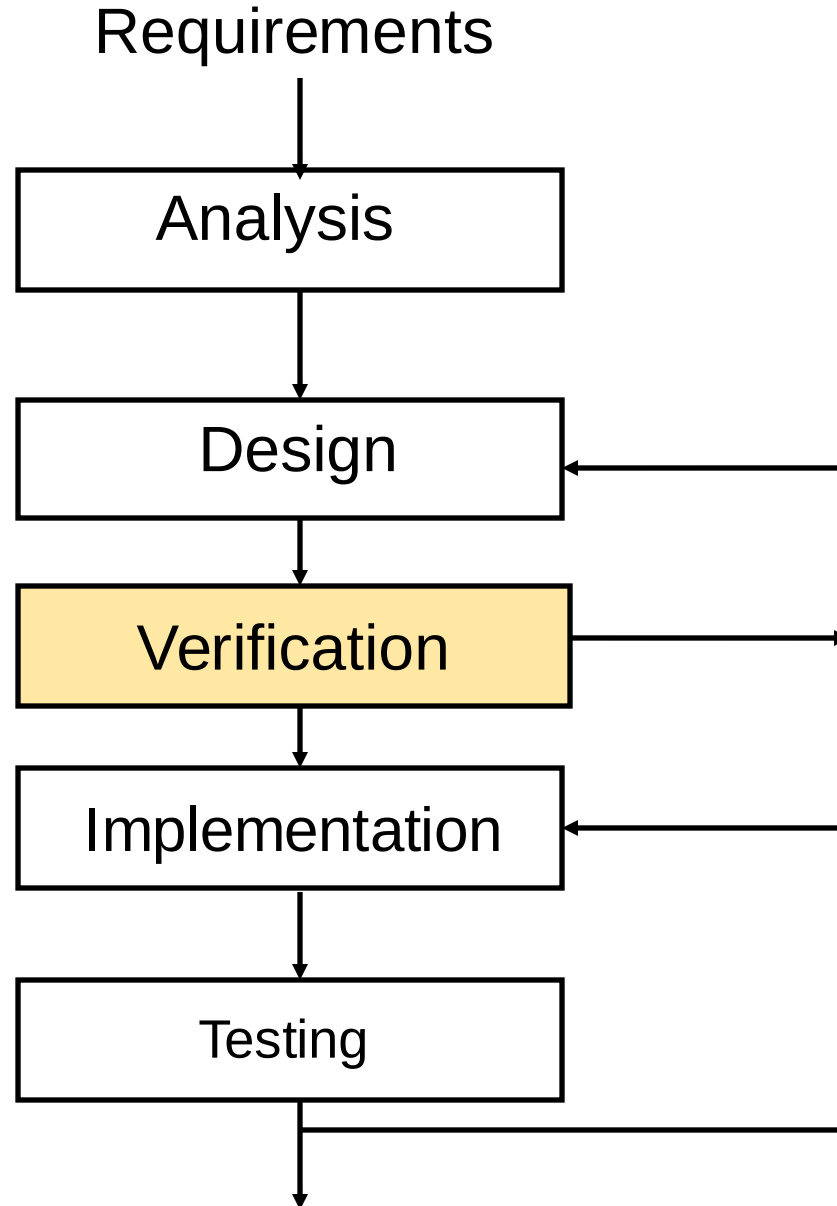
# Model-based Development

- 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)
- **Lustre, Heptagon**, SCADE, Esterel Studio (Esterel Technologies)
- Stateflow and Simulink (Mathworks)

# Modeling Languages

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

# Modeling Languages

- Choice of languages depends on 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

# Summary

- **Embedded Systems are complex**
  - Correct functioning is crucial
  - They are reactive systems (RS)
  - They are Concurrent
  - They are Realtime
  - With stringent resource constraints
- **System development methodology needed**
  - To model reactive systems
  - Derive implementation from model
  - Verification capability is important
- **New System building paradigm:**
  - Model – Verify – Debug – CodeGenerate

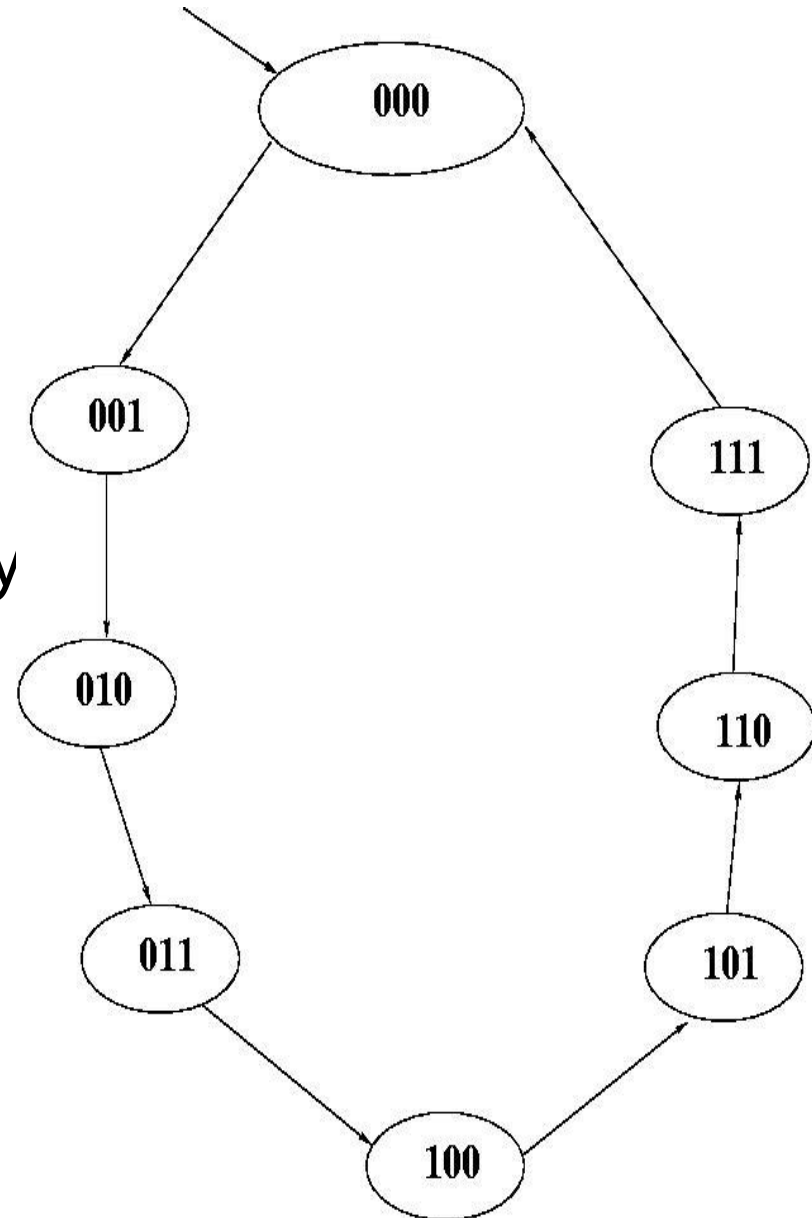
# Models and Tools for Embedded Systems

# 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 (Hardware model)



# State Machine Model

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

# Precise Definition

$\langle Q, q_0, S, T \rangle$

- $Q$  – A finite no. of state names
  - $q_0$  – Initial state
  - $S$  – Edge alphabet
  - $T$  – edge function or relation
- 
- **Abstract labels to concrete event, condition and action**

# Semantics

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

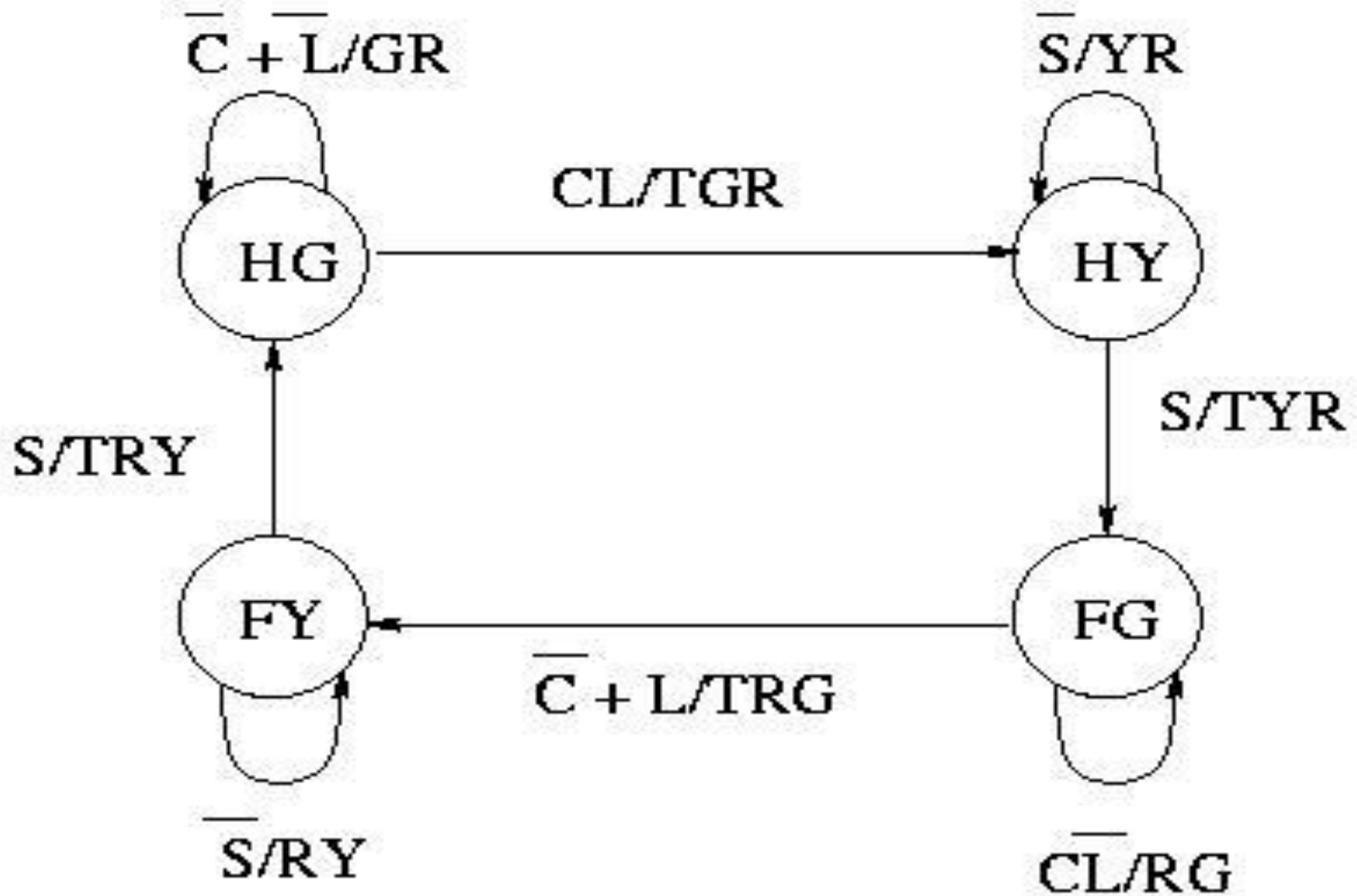
- FSM models can be refined to different impl.
  - Both HW and SW implementation
  - Exploring alternate implementations
  - For performance and other considerations
- Automatic code generation
  - Preferable over hand generated code
  - Quality is high and uniform

# Another Example

## **A Traffic Light Controller**

- Traffic light at intersection of Highway & 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 hway green & farm road red

# State Machine



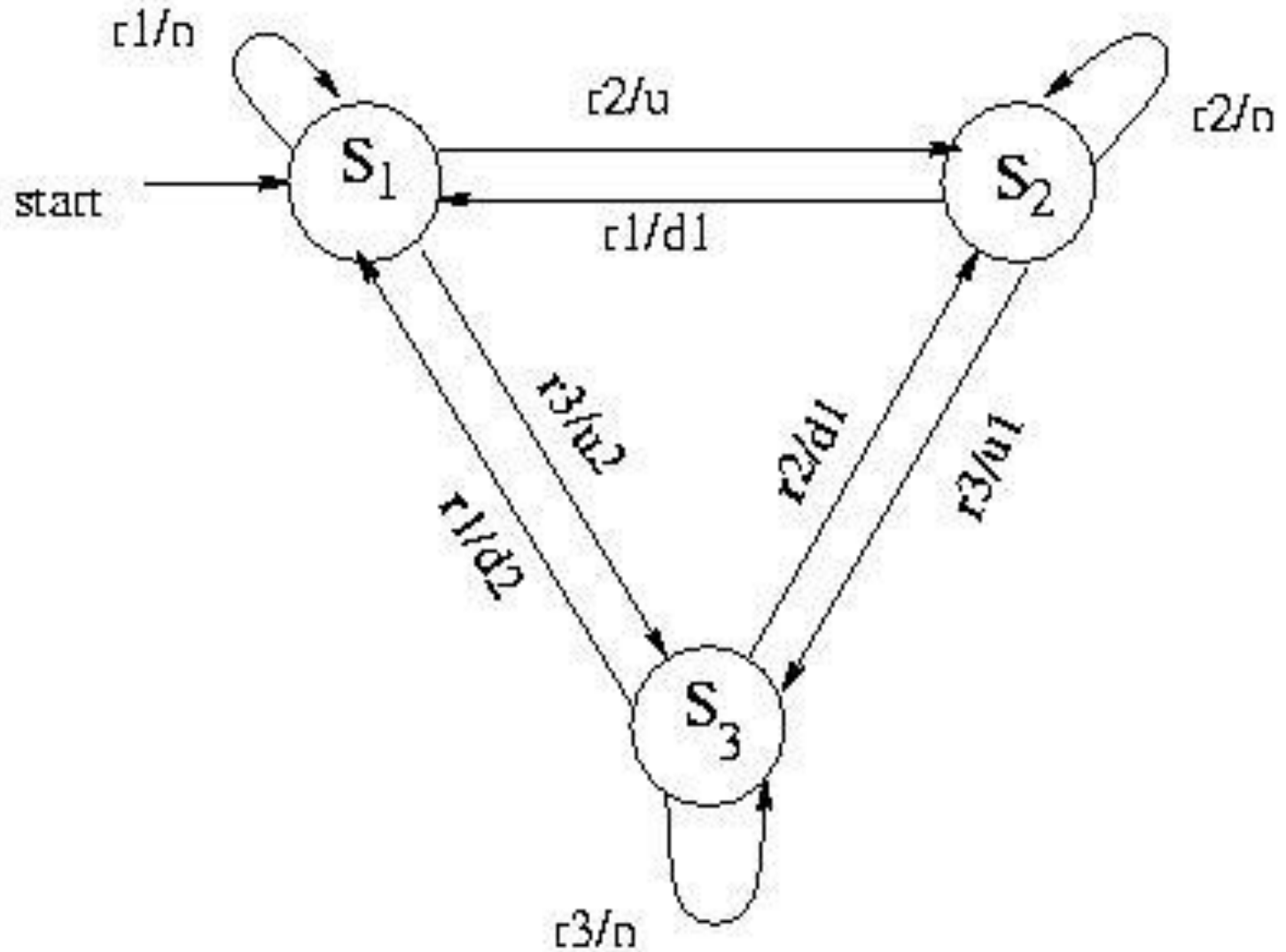
# Another Example

## A Simple Lift Controller

### 3-floor lift

- Lift can be in any floor
  - $S_i$  - on floor  $i$
- Request can come from any floor
  - $r_i$  - request from floor  $i$
- Lift can be asked to move up or down
  - $u_j, d_j$  - up/down to  $j$ th floor

# FSM model



# Nondeterminism

- Suppose lift is in floor 2 (State  $S_2$ )
- What is the next state 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
- Models can be nondeterministic but not real lifts!

# Nondeterminism

- Models focus attention on a particular aspect
- The lift model focused 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 **combining different models**

# 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;};
    break; }; /* end of switch */
req = read_req(); } /* end of while */
```

# Suitability of C

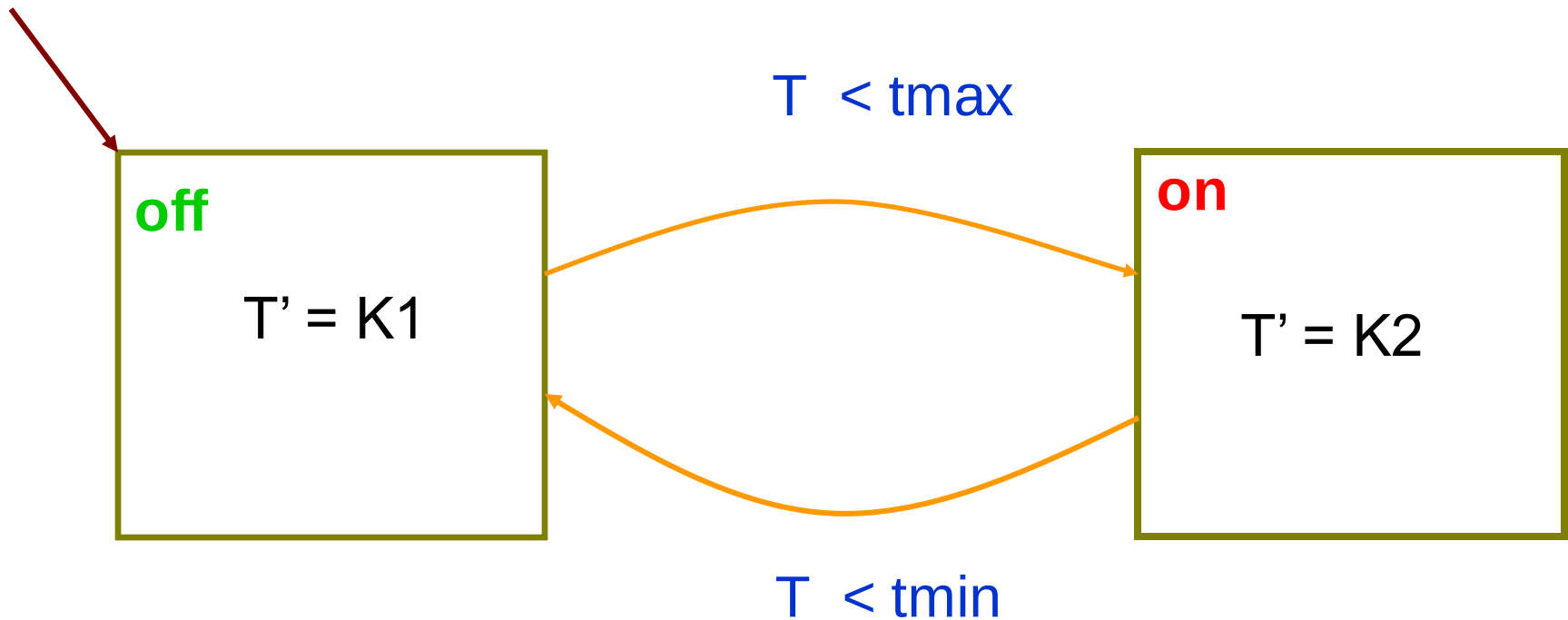
- C not natural for such applications
- Various problems
  - Events and states all modeled as variables
  - Not natural for event oriented embedded applications
  - States are implicit (control points decide states)
  - No abstract description possible
  - Commitment to details at an early stage
  -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

- 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 & combining dif. state machines

# References (Old work)

- 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
- Edward Lee and Sanjit Seshia, [Introduction to Embedded Systems](#), MIT Press 2017.