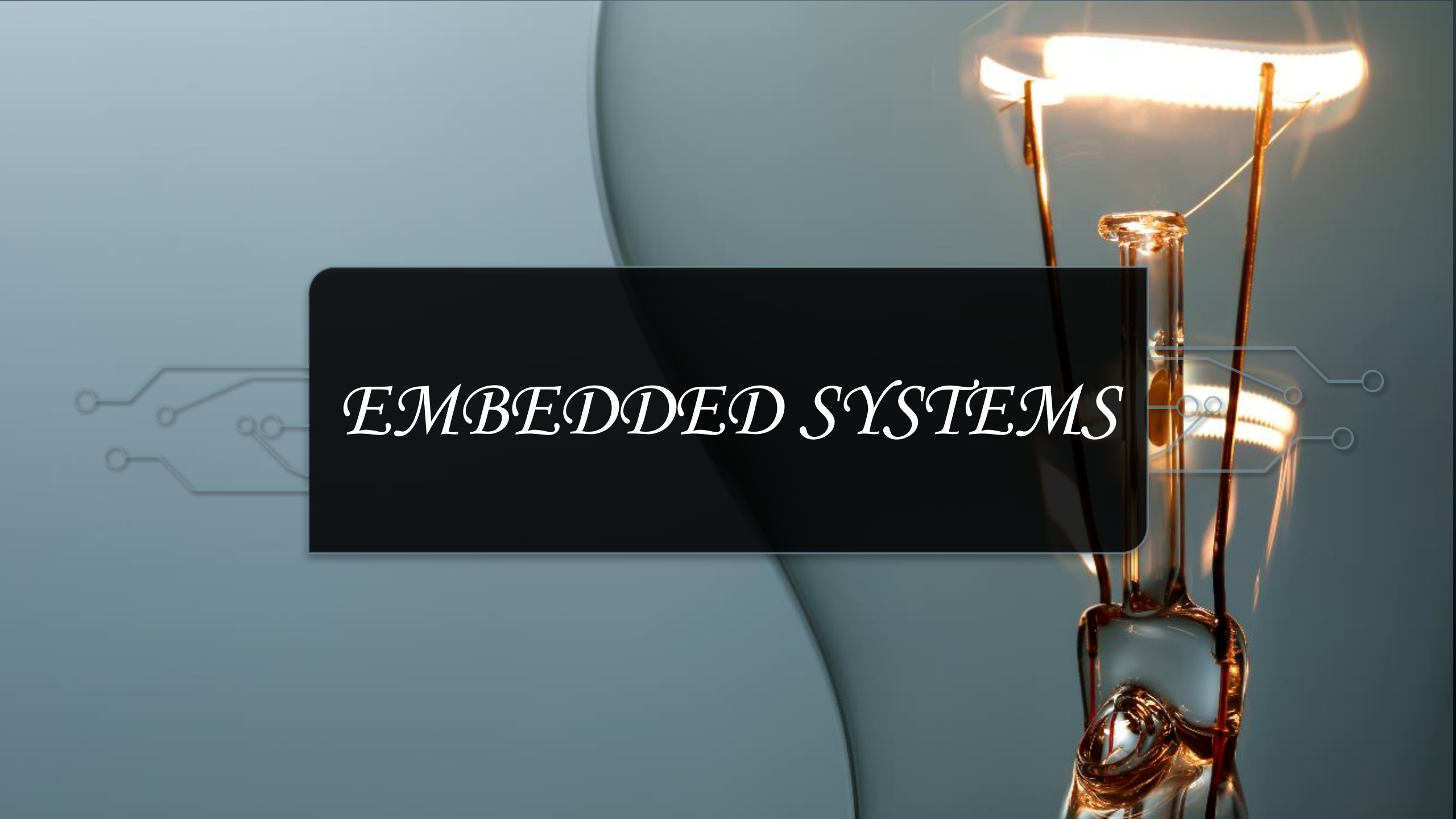
EMBEDDED SYSTEMS



Embedded Systems Module 1

- Introduction to Embedded System
- Application Areas
- Growing importance of embedded systems
- Models of Computation
- State Charts
- General Language Characteristics

History of Embedded Systems

- Early 1980's large mainframe computers and huge tape drivers
- Later due to miniaturization, evolved the "information processing with personal computers"
- Mark Weiser termed ubiquitous computing
- Ubiquitous computing is to have computing anytime and anywhere
- Ubiquitous computing: focus on long term goal of providing information anytime and anywhere
- He also termed it as computers are going to be invisible computers

Introduction to Embedded Systems

What is a System?

- *In simple, a system is a set of interrelated parts/component which are designed/developed to perform a common tasks or to do some specific work for which it has been created.*

What does Embedded mean?

- *Embedded means something which is integrated or attached to another thing.*

What is a Embedded System?

- An Embedded System is an integrated system which is formed as an combination of computer hardware and software for a specific function.
- Miniaturization also enabled the integration of information processing and the environment using computers. This type of information processing is called as embedded system.
- It can be said as a dedicated computer system which has been developed for some particular reason. But it is not our traditional computer system or general-purpose computers, these are the Embedded systems which may work independently or attached to a larger system to work on few specific functions.
- These embedded systems can work without human intervention or with little human intervention.

Cyber-Physical System (CPS)

Embedded System

Physical Environment

- *Embedded systems (ES) are information processing systems embedded into enclosing products.*
- *ES – the information processing part*
- *Cyber-Physical Systems (CPS) are integrations of computation and physical processes.*
- *Cyber-Physical Systems (CPS) refer to next generation embedded ICT systems that are interconnected and collaborating including through the Internet of Things, and providing citizens and businesses with a wide range of innovative applications and services*
- *The term Internet of Things "describes the pervasive presence of a variety of devices — such as sensors, actuators, and mobile phones through which unique addressing schemes, are able to interact and cooperate with each*

Important Characteristics of an Embedded System

- *Performs specific task:* Embedded systems performs some specific function or tasks.
- *Low Cost:* The price of an embedded system is not so expensive.
- *Time Specific:* It performs the tasks with in a certain time frame.
- *Low Power:* Embedded Systems don't require much power to operate.
- *High Efficiency:* The efficiency level of embedded systems are so high.
- *Minimal User interface:* These systems require less user interface and easy to use.
- *Less Human intervention:* Embedded systems require no human

Important Characteristics of an Embedded System

- *Highly Stable:* Embedded systems not change frequently mostly fixed maintaining stability.
- *High Reliability:* Embedded systems are reliable they perform the tasks consistently well.
- *Use microprocessors or micro controllers:* Embedded systems use microprocessors or micro controllers to design and use limited memory.
- *Manufacturable:* The majority of embedded systems are compact and affordable to manufacture. They are based on the size and low

Advantages of Embedded System

- *Small size.*
- *Enhanced real-time performance.*
- *Easily customizable for a specific application.*

Disadvantages of Embedded System

- *Limited memory.*
- *Time-consuming design process.*

- *Computing devices are termed into pervasive computing and ambient intelligence*
- *Pervasive computing : focus more on physical, practical aspects and altering the already existing technology*
- *Ambient Intelligence: based on the communication technology in future homes and smart buildings*

-Application Areas

-Growing importance of ES

1. Mechanical Engineering
2. Robotics
3. Power Engineering and the smart grid
4. Civil Engineering
5. Disaster Recovery
6. Smart Building
7. Agricultural Engineering
8. Health sector and medical engineering
9. Scientific experiments
10. Public Safety
11. Military applications
12. Telecommunication
13. Consumer electronics
14. Transportation Mobility

Three main components of Embedded systems are:

1. Hardware
2. Software
3. Firmware

Digital watches

Cameras

Industrial machines

Electronic Calculators

Automobiles

Medical Equipment

Washing Machine

Toys

Televisions

Digital phones

Laser Printer

Requirements of an Embedded System Design

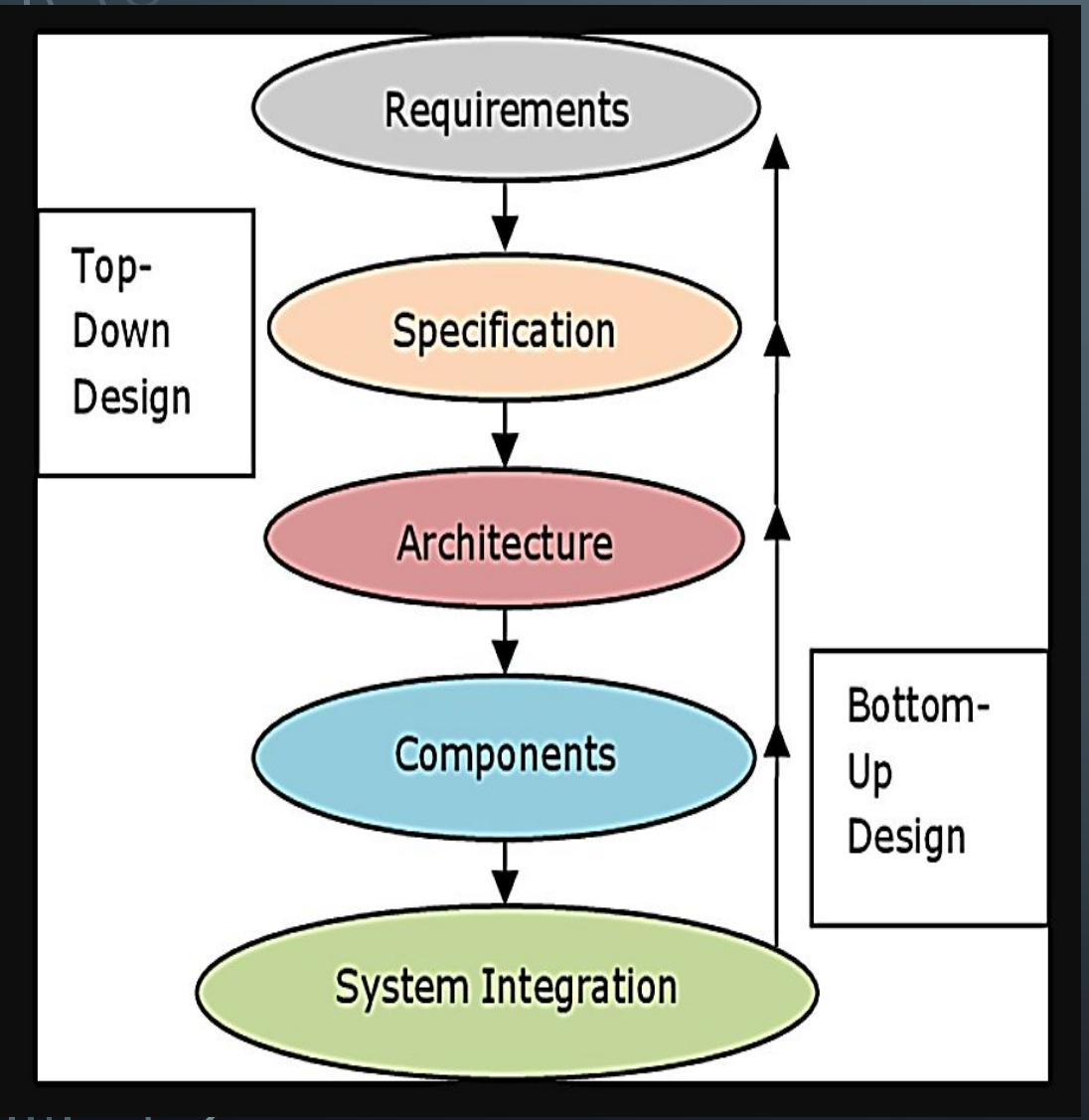
1. *Introduction to Embedded System Design Process*
2. *Major steps in the embedded system design process*
3. *Requirements of an embedded system in detail*
4. *Typical Non-functional requirements include*
5. *Mock-up*
6. *Sample Requirements Form*

Introduction to Embedded System Design Process

A design methodology is important for three reasons.

- **First,** it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing performance or performing functional tests.
- **Second,** it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and emits a completed design would be a daunting task, but by first breaking the process into manageable steps, we can work on automating (or at least semi automating) the steps one at a time.
- **Third,** a design methodology makes it much easier for members of a design team to communicate.

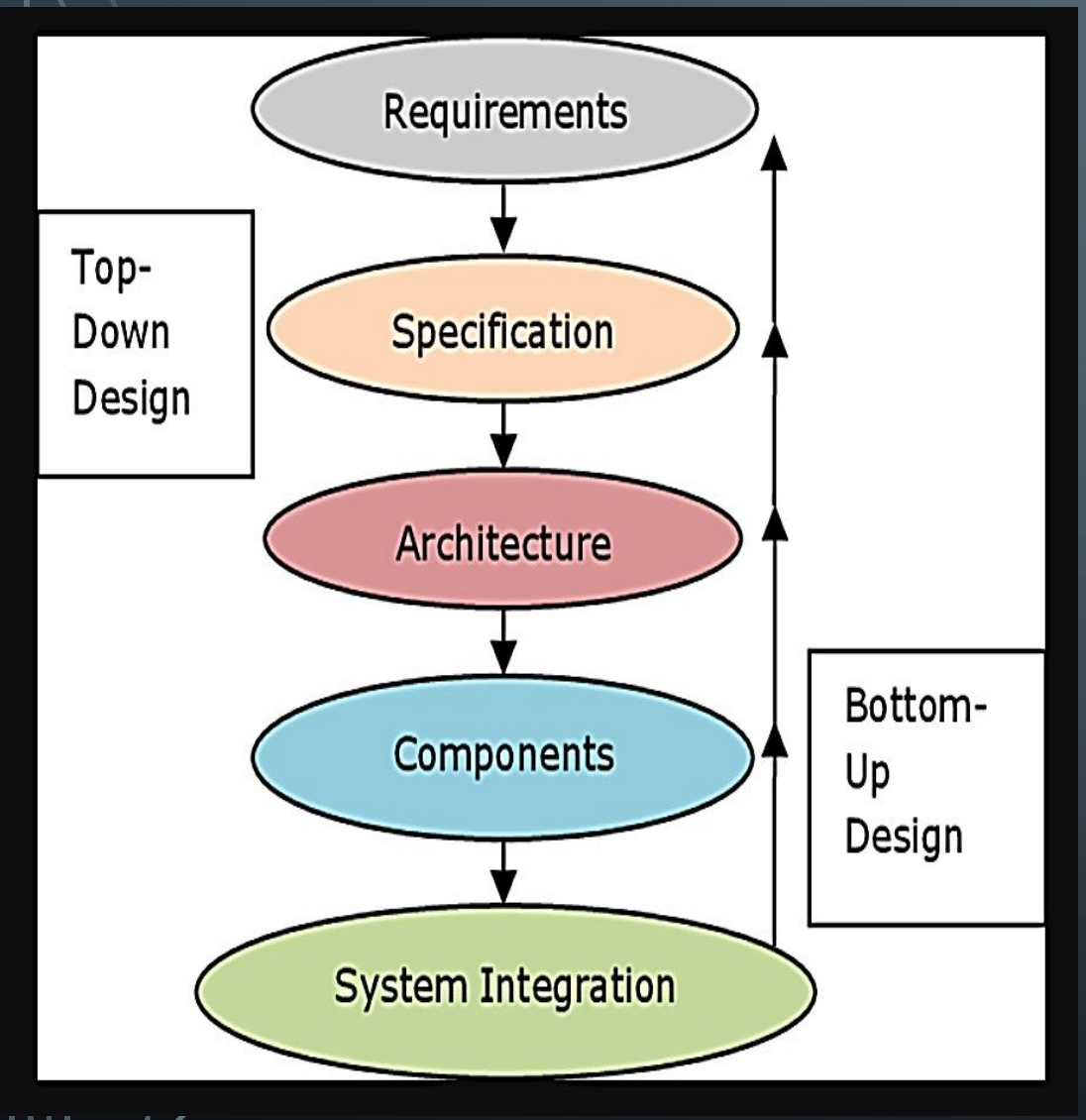
Major steps in the embedded system design process



Requirements: before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components

Specification: we create a more detailed description of what we want. But the specification states only how the system behaves, not how it is built.

Major steps in the embedded system design process



Architecture: The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components.

Components: Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need.

System Integration: Final step in the design process in which plugging everything together is performed. Bugs are typically found during system integration, and good planning can help us find the bugs quickly.

Requirements of an embedded system in detail

Basically there are two types of requirements: Functional and Non- Functional requirements

A functional requirement defines a system or its component.

A non-functional requirement defines the quality attribute of a software system. It specifies "What should the software system do?" It places constraints on "How should the software system fulfill the functional requirements?"

Example: A system loads a webpage when someone clicks on a button is a functional requirement. But The related non-functional requirement specifies how fast the webpage must load.

The Bottom Line. Systems need both functional requirements and non functional requirements to be usable. Functional requirements define how the system must work and non functional requirements detail how it should perform. Without functional requirements the system will not work.

Typical Non-functional requirements include

1. Performance:

- The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost.
- Performance may be a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed.

2. Cost:

- The target cost or purchase price for the system is almost always a consideration.
- Cost typically has two major components: manufacturing cost and nonrecurring engineering (NRE) costs
- Manufacturing cost includes the cost of components and assembly
- Non-Recurring engineering (NRE) costs include the personnel and other costs of designing the system.

Typical Non-functional requirements include

3. Physical size and weight:

- The physical aspects of the final system can vary greatly depending upon the application.
- An industrial control system for a assembly line may be designed to fit into a standard-size rack with no strict limitations on weight.
- A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.

4. Power consumption:

- Power is important in battery-powered systems and is often important in other applications as well.
- Power can be specified in the requirements stage in terms of battery life—the customer is unlikely to be able to describe the allowable wattage.

Mock-up

- Validating a set of requirements is ultimately a psychological task since it requires understanding both what people want and how they communicate those needs.
- One way to refine at least the user interface portion of a system's requirements is to build a mock-up.
- The mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation.
- But it should give the customer a good idea of how the system will be used and how the user can react to it.

Sample Requirements Form

Name

Purpose

Inputs

Outputs

Functions

Performance

Manufacturing Cost

Power

Physical size and Weight

- Sample requirements form that can be filled out at the start of the project.
- We can use the form as a checklist in considering the basic characteristics of the system.

1. Name:

- Giving a name to the project not only simplifies talking about it to other people
- but can also crystallize the purpose of the machine.

2. Purpose:

- This should be a brief one- or two-line description of what the system is supposed to do.
- If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.

3. Inputs and outputs:

These two entries are more complex than they seem.

The inputs and outputs to the system encompass a wealth of detail:

— Types of data: Analog electronic signals? Digital data? Mechanical inputs?

— Data characteristics: Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?

— Types of I/O devices: Buttons? Analog/digital converters? Video display?

4.Functions:

- *This is a more detailed description of what the system does.*
- *A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?*

5. Performance:

- *Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world.*
- *In most of these cases, the computations must be performed within a certain time frame.*
- *It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.*

6. Manufacturing cost:

- *This includes primarily the cost of the hardware components.*
- *Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range.*

7. Power:

- *Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way.*
- *Battery-powered machines must be much more careful about how they spend energy.*

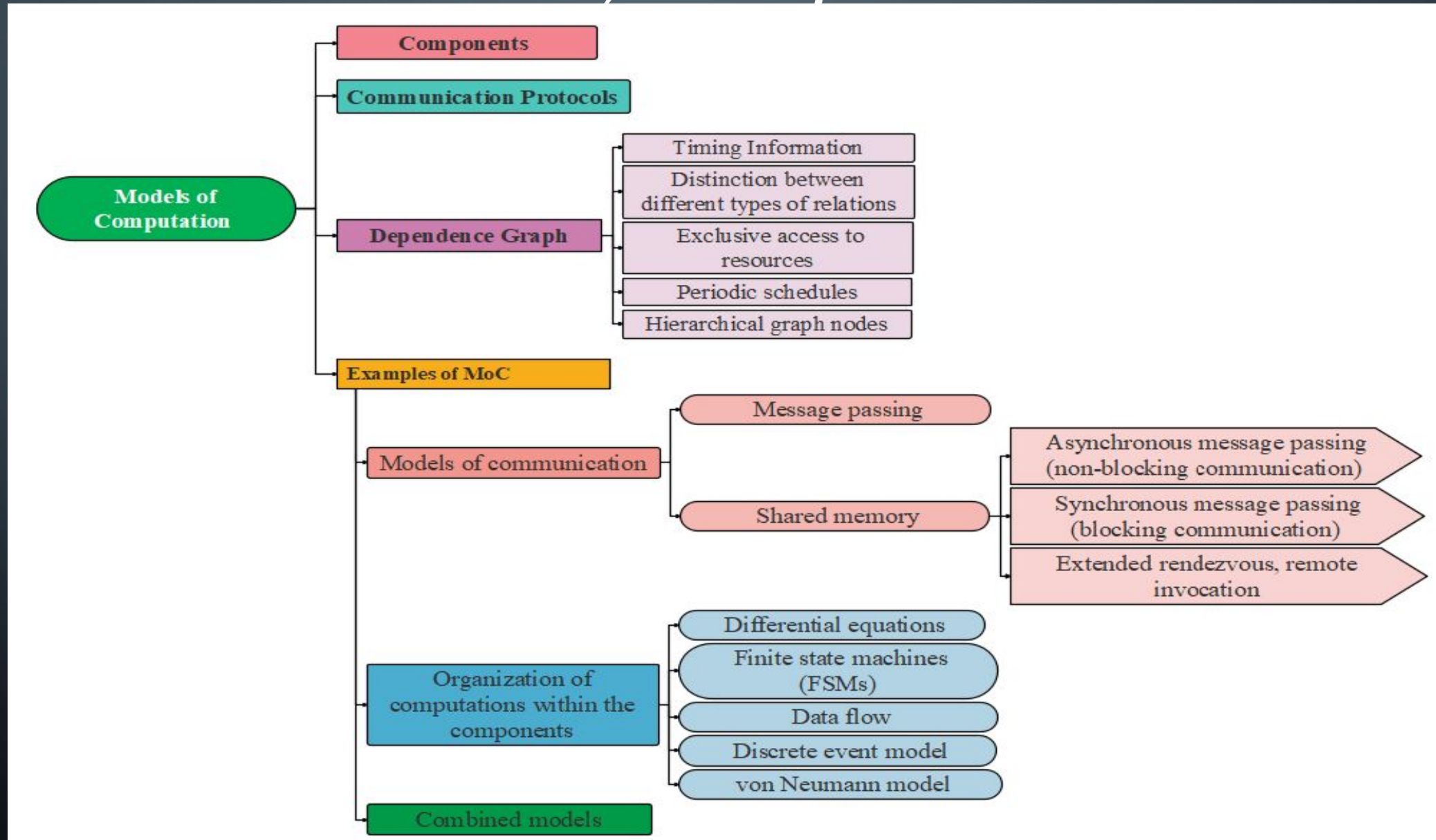
8. Physical size and weight:

- *You should give some indication of the physical size of the system to help guide certain architectural decisions.*

Example of a Sample Requirement Form

Name	• GPS Moving Map
Purpose	• Consumer-grade moving map for driving use
Inputs	• Power button, two control buttons
Outputs	• Back-lit LCD display 400*600
Functions	• Uses 5 receiver GPS system; 3 user selectable resolution; always display current latitude and longitude
Performance	• Updates screen within 0.25 seconds upon movement
Manufacturing Cost	• $30
Power	• 100mW
Physical Size And Weight	• No more than 2''*6", 12 ounces

Models of computation



Models of computation

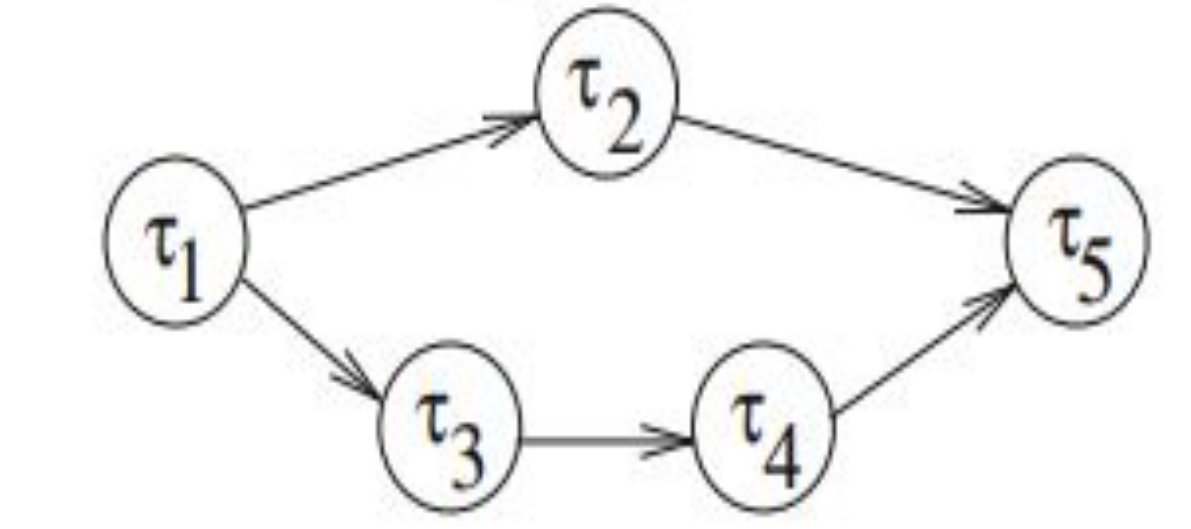
MoCs:

Models of computation (MoCs) describe the mechanism assumed for performing computations.

Components *and the organization of computations in components: Procedures, processes, functions, and finite state machines are possible components.*

Communication protocols *These protocols describe methods for communication between components. Asynchronous message passing & rendezvous-based communication are examples of communication protocols.*

Dependence Graph



τ is the set of vertices or nodes

τ1 - immediate predecessor of τ2

τ2 - immediate successor

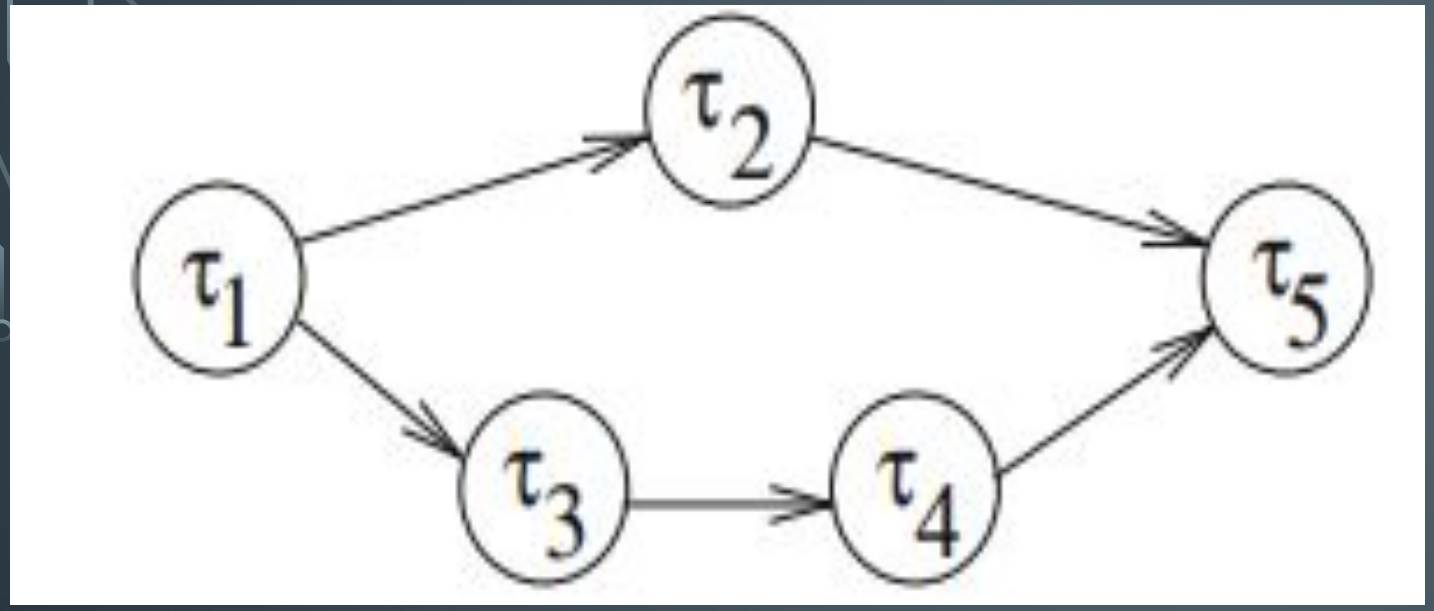
A dependence graph is a directed graph G = (τ, E), where τ is the set of vertices or nodes and E is the set of edges.

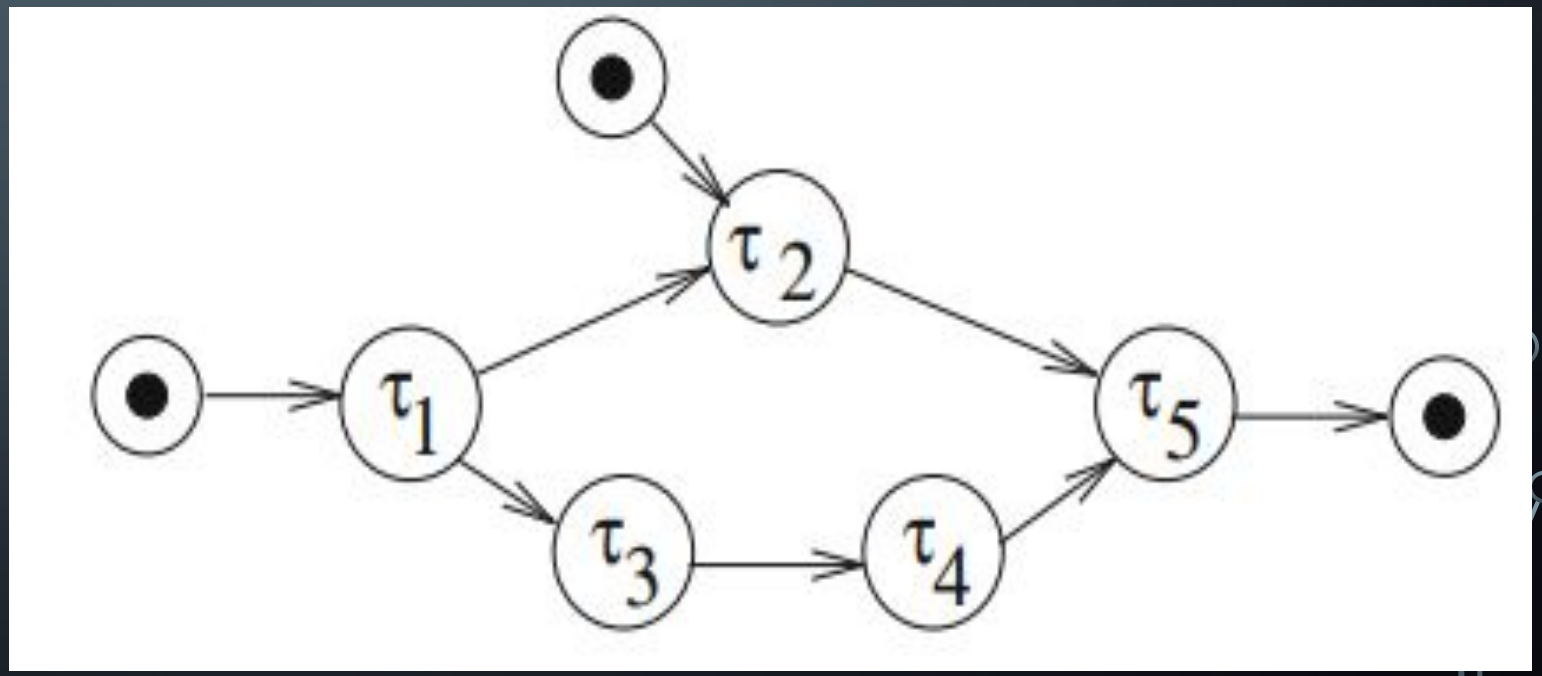
- *Relations between components can be captured in graphs.*
- *In such graphs, we will refer to the computations also as processes or tasks. Accordingly, relations between these will be captured by task graphs and process networks.*
- *Nodes in the graph represent components performing computations. Computations map input data streams to output data streams.*
- *Computations are sometimes implemented in high-level programming languages. Typical computations contain (possibly non-terminating) iterations.*

- *Task graphs may include the following extensions of dependence graphs*
- *Timing information*
- *Distinction between different types of relations between computations*
- *Exclusive access to resources*
- *Periodic schedules*
- *Hierarchical graph nodes*

- *Timing information*
  - *Tasks may have arrival times, deadlines, periods, and execution times.*
  - *In order to show them graphically, it may be useful to include this information in the graphs.*

- *Distinction between different types of relations between computations*
- *Priority relations are just model constraints for possible execution sequences*
- *It may be useful to distinguish between constraints for scheduling and communication between*


τ2
τ1
τ5
τ3
τ4


τ2
τ1
τ5
τ3
τ4

- *Exclusive access to resources*
- *Computations may be requesting exclusive access to some resource, for example, to some input/output device or some communication area in memory*
- *Information about necessary exclusive access should be taken into account during scheduling.*

- *Periodic schedule*
- *Many computations, especially in digital signal processing, are periodic.*
- *This means that we must distinguish more carefully between a task and its execution.*
- *Graphs for such schedules are infinite. A graph*

- *Hierarchical graph nodes*
- *Granularity:*
  - *On the one hand, specified computations may be quite involved and contain thousands of lines of program code.*
  - *On the other hand, programs can be split into small pieces of code so that in the extreme case, each of the nodes corresponds only to a single operation. The graph*

Examples of MoC

A. Models of communication

1. Shared Memory:

- For shared memory, communication is performed by accesses to the same memory from all components.
- Access to shared memory should be protected, unless access is restricted to reads.
- If writes are involved, exclusive access to the memory must be guaranteed while components are accessing shared memories.
- Segments of program code, for which exclusive access must be guaranteed, are called critical sections
- Shared memory-based communication can be fast but is difficult to implement in

2. Message passing

- In this case, messages are sent and received. Message passing can be implemented easily even if no common memory is available.
- However, message passing is generally slower than shared memory-based communication.
- Three kinds of message passing

1. Asynchronous message passing, also called non-blocking communication
2. Synchronous message passing or blocking communication, rendezvous based communication

1. *Asynchronous message passing, also called non-blocking communication*

- *In asynchronous message passing, components communicate by sending messages through channels which can buffer the messages.*
- *The sender does not need to wait for the recipient to be ready to receive the message.*
- *In real life, this corresponds to sending a letter or an e-mail.*
- *A potential problem is the fact that messages must be stored an that message buffers can overflow.*
- *There are variations of this scheme, including communicating finite*

2. Synchronous message passing or blocking communication, rendezvous based communication

- In synchronous message passing, available components communicate in atomic, instantaneous actions called rendezvous.
- The component reaching the point of communication first has to wait until the partner has also reached its point of communication.
- In real life, this corresponds to physical meetings or phone calls. There is no risk of overflows, but performance may suffer.

3. Extended rendezvous, remote invocation

- In this case, the sender is allowed to continue only after an acknowledgment has been received from the recipient.
- The recipient does not have to send this acknowledgment immediately after receiving the message but can do some preliminary checking before actually sending the acknowledgment.

B. Organization of computations within the components

- ***Differential equations:*** *Differential equations are capable of modeling analog circuits and physical systems. Hence, they can find applications in cyber-physical system modeling.*
- ***Finite state machines (FSMs):*** *This model is based on the notion of a finite set of states, inputs, outputs, and transitions between states. Several of these machines may need to communicate, forming so-called communicating finite state machines (CFSMs).*
- ***Data flow:*** *In the data-flow model, the availability of data triggers*

- *Discrete event model:* In this model, there are events carrying a totally ordered time stamp, indicating the time at which the event occurs. Discrete event simulators typically contain a global event queue sorted by time. Entries from this queue are processed according to this order. The disadvantage is that this model relies on a global notion of event queues, making it difficult to map the semantic model onto parallel implementations. Examples include VHDL, SystemC and Verilog.
- *Von Neumann model:* This model is based on the sequential execution of sequences of primitive computations

C. Combined models

- *Actual languages are typically combining a certain model of communication with an organization of computations within components.*
- *For example, State Charts combines finite state machines with shared memories.*
- *SDL (specification and description language) combines finite state machines with asynchronous message passing.*

General Language Characteristics

Communication/ organization of components	Shared memory	Message passing	
		Synchronous	Asynchronous
Undefined components	Plain text or graphics, use cases		
		(Message) sequence charts	
Differential equations	Modelica, Simulink®, VHDL-AMS		
Communicating finite state machines (CFSMs)	StateCharts		SDL
Data flow	Scoreboarding, Tomasulo algorithm		Kahn networks SDF
Petri nets		C/E nets, P/T nets, ...	
Discrete event (DE) model[a]	VHDL, Verilog SystemC	(Only experimental systems) Distributed DE in Ptolemy	
von Neumann model	C, C++, Java	C, C++, Java, ... with libraries	
		CSP, Ada	

State Charts: Implicit Shared Memory Communication

- The State Charts language describes extended FSMs.
- Due to this, they can be used for modeling state-oriented behavior. The key extension is hierarchy.
- Hierarchy is introduced by means of superstates, States comprising other states are called superstates.
- States included in superstates are called substates of the superstates.

- *The State Charts diagram in Fig. 2 is a hierarchical version of the diagram in Fig. 1.*
- *Superstate S includes states A, B, C, D, and E. Suppose the FSM is in state Z (Z will also be called an active state).*
- *Now, if input m is applied to the FSM, then A and S will be the new active states.*
- *If the FSM is in S and input k is applied, then Z will be the new active state, regardless. Whenever a substate of some superstate is active, the superstate is active as well.*

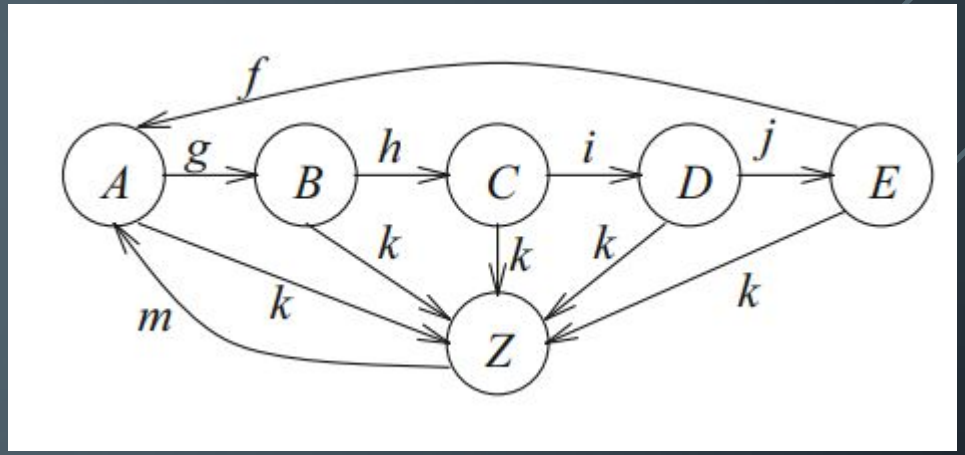


Fig. 1

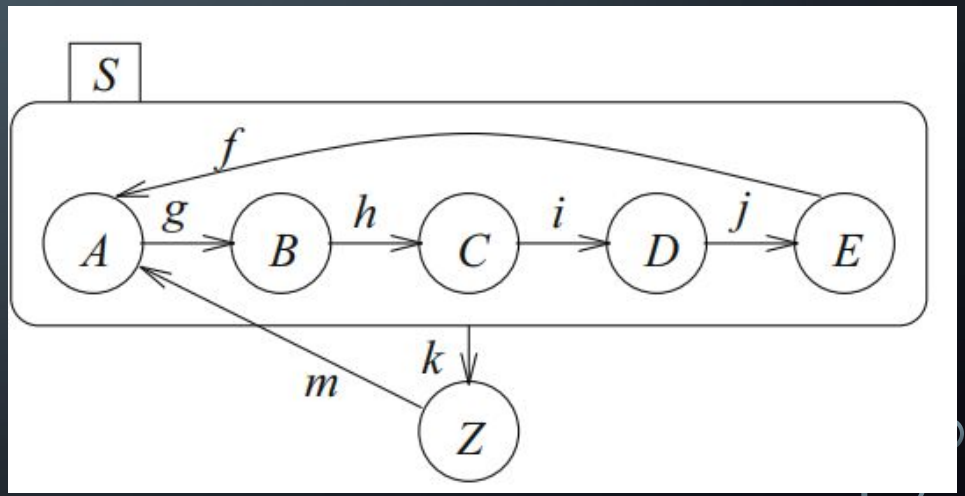
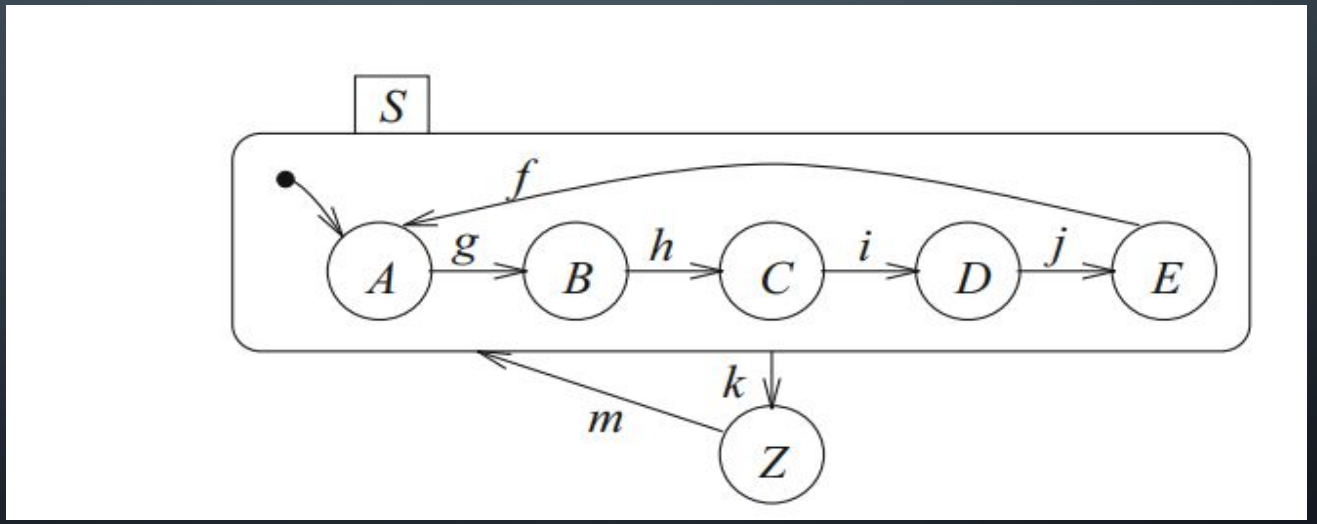


Fig. 2

- *States which are not composed of other states are called basic states. The FSM of Fig. 2 can only be in one of the substates of substate S at any time.*
- *Superstates of this type are called OR-superstates. k might correspond to an exception for which state S has to be left*
- *Superstate S is called an OR-superstate if the system comprising S is in exactly one substate of S whenever it is in S.*
- *State Charts allows hierarchical descriptions of systems in which a system description comprises descriptions of subsystems which, in turn, may contain descriptions of subsystems.*
- *The hierarchy of the entire system can be represented by a tree. The root of the tree corresponds to the system as a whole, and all inner nodes correspond to hierarchical descriptions (called super-nodes in State Charts).*
- *The leaves of the hierarchy are non-hierarchical descriptions.*

- *Default state mechanism can be used in superstates to indicate the particular substates that will become active if the superstates become active.*
- *In diagrams, default states are identified by edges starting at small filled circles.*



- *Another mechanism for specifying next states is the history mechanism. With this mechanism, it is possible to return to the last substate that was active before a superstate was left. The history mechanism is symbolized by a circle containing the letter H. D*
- *Consider the state diagram in below Fig a. The behavior of the FSM is now somewhat different.*
- *If we input m while the system is in Z, then the FSM will enter A if this is the very first time we enter S, and otherwise it will enter the last state that we were in before leaving S. This mechanism has many applications.*
- *For example, if k denotes an exception, we could use input m to return to the state we were in before the exception. States A, B, C, D, and E could also call Z like a procedure. After completing "procedure" Z, we would return to the calling state.*
- *In this way, we are adding elements of programming languages to State Charts. Figure a can also be redrawn as shown in Fig. b. In this case, the symbols for the default and the history*

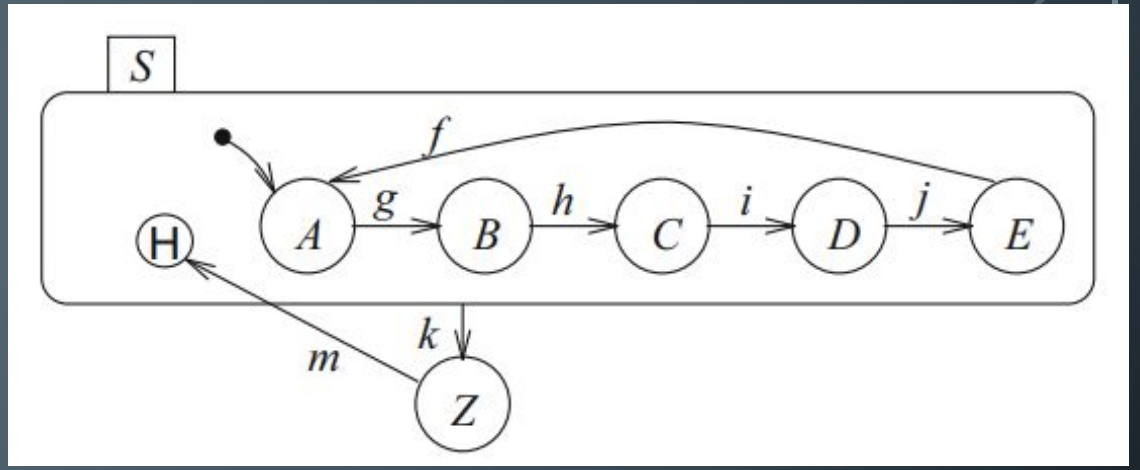


Fig a

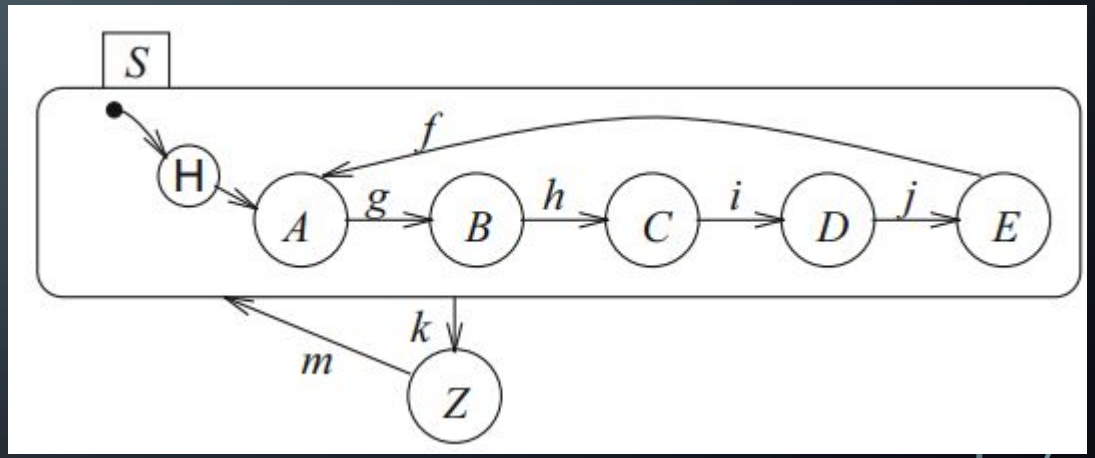
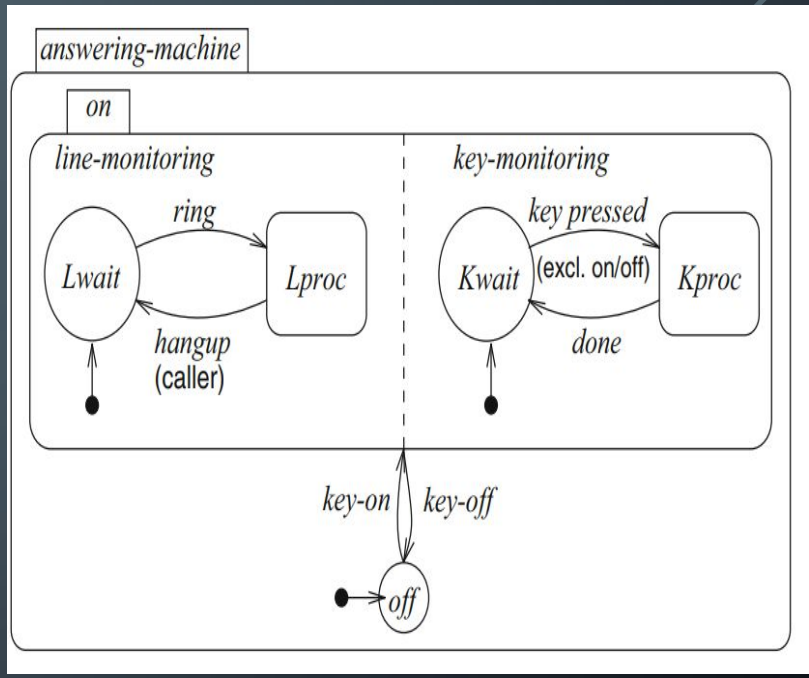
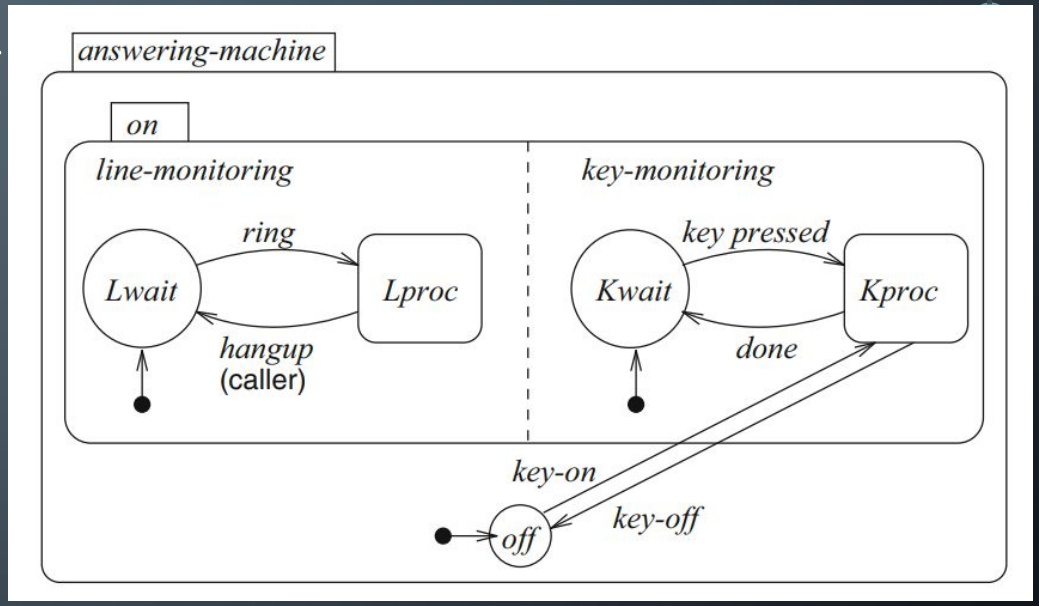


Fig b

- *Superstates S are called AND-superstates if the system containing S will be in all of the substates of S whenever it is in S. An AND-superstate is included in the answering machine example shown in Figure.*
- *An answering machine normally performs two tasks concurrently: it is monitoring the line for incoming calls and the keys for user input.*
- *In Fig. the corresponding states are called Lwait and Kwait. Incoming calls are processed in state Lproc, while the response to pressed keys is generated in state Kproc.*
- *State Lproc is left whenever the caller hangs up the phone. Returning to state Lwait due to call termination by the owner is not modeled.*
- *Hence, this model provides no protection against stalking.*
- *For the time being, we assume that the on/off switch (generating events key-off and key-on) is decoded separately and pushing it does not result in entering Kproc. If the machine is switched off, the line monitoring state and the key monitoring state are left and reentered only if the machine is switched on.*
- *At that time, default states Lwait and Kwait are entered. While switched on, the machine will always be in the line monitoring state as well as in the key monitoring state. For*

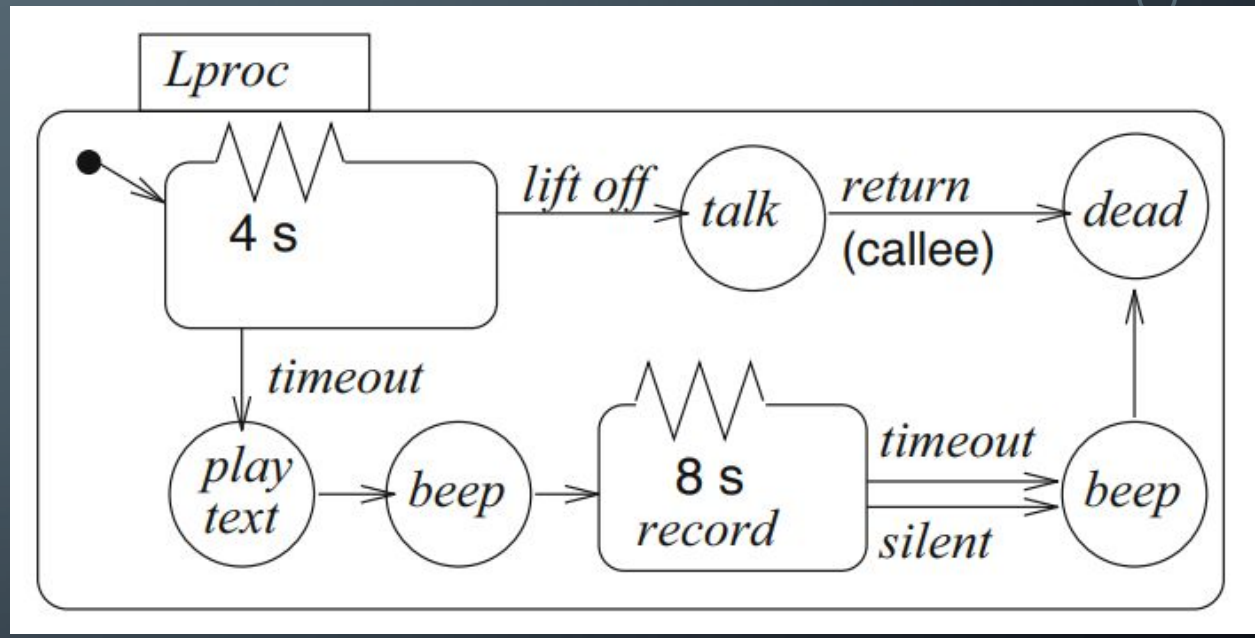


- *Let us modify our answering machine such that the on/off switch, like all other switches, is decoded in state Kproc as Fig below. If pushing that key is detected in Kwait, transitions are assumed first into state Kproc and then into the off state.*
- *The second transition results in leaving the linemonitoring state as well. Switching the machine on again results in also entering the line-monitoring state.*
- *AND-superstates provide the key mechanism for describing concurrency in StateCharts. Each substate can be considered a state machine by itself.*
- *These machines are communicating with each other, forming communicating finite state machines (CFSMs).*



TIMERS

- StateCharts also provides timers.
- After the system has been in the state containing the timer for the specified time, a timeout will occur, and the system will leave the specified state.
- Fig shows a possible behavior for that state.
- Due to the exception-like transition for hangups by the caller in previous fig, state Lproc is terminated whenever the caller hangs up.
- For hangups (returns) by the callee, the design of state Lproc results in an inconvenience: if the callee hangs up the phone first, the telephone will be dead (and quiet) until the caller has also hung up the phone.



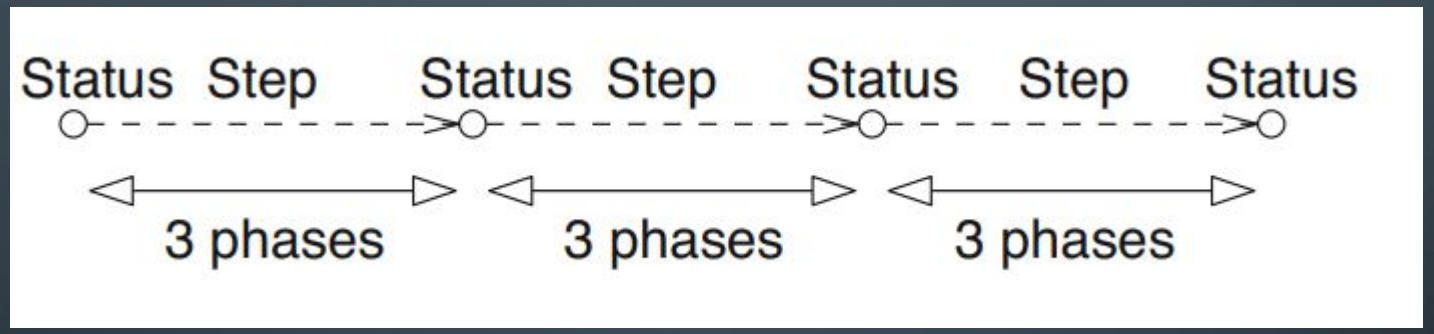
Edge Labels and Statemate Semantics

- *Generated outputs can be specified using edge labels.*
- *The general form of an edge label is "event [condition]/reaction." All three label parts are optional.*
- *The reaction part describes the reaction of the FSM to a state transition. Possible reactions include the generation of events and assignments to variables.*
- *The condition part implies a test of the values of variables or a test of the current state of the system.*
- *The event part refers to a test of current events. Events can be generated either internally or externally.*
- *Internal events are generated as a result of some transition and are described in reaction parts.*
- *External events are usually described in the model environment*

Examples:

- *on-key*/*on*:=1 (Event test and variable assignment),
- [*on*=1] (Condition test for a variable value),
- *off-key* [not in *Lproc*]/*on*:=0 (Event test, condition test for a state, variable assignment. The assignment is performed if the event has occurred and the condition is true).

- *The semantics of edge labels can only be explained in the context of the semantics of StateMate.*
- *Steps are assumed to be executed each time events or variables have changed. The set of all values of variables, together with the set of events generated (and the current time), is defined as the status13 of a StateMate model. After executing the third phase, a new status is obtained.*



- *The visibility of events is limited to the step following the one in which they are generated. Thus, events behave like single bit values which are stored in permanently enabled registers at one clock transition and have an effect on the values stored at the next clock transition. They do not live forever.*

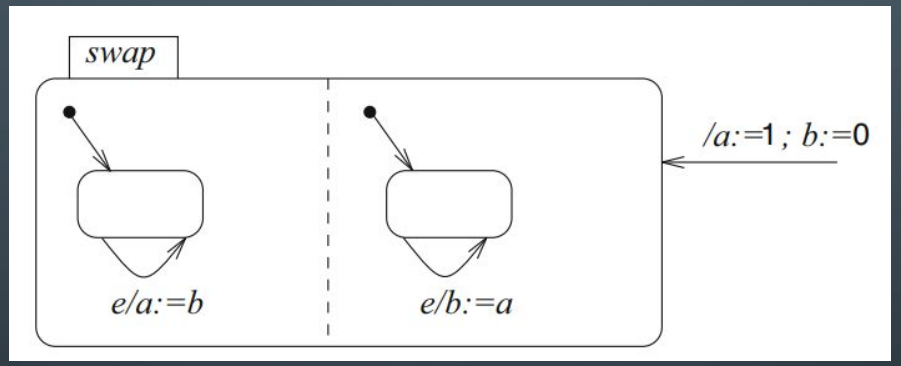
- *Each step consists of three phases:*

1. In the first phase, the impact of external changes on conditions and events is evaluated. This includes the evaluation of functions which depend on external events. This phase does not include any state changes.

2. The next phase is to calculate the set of transitions that should be made in the current step. Variable assignments are evaluated, but the new values are only assigned to temporary variables.

3. In the third phase, state transitions become effective and variables obtain their new values

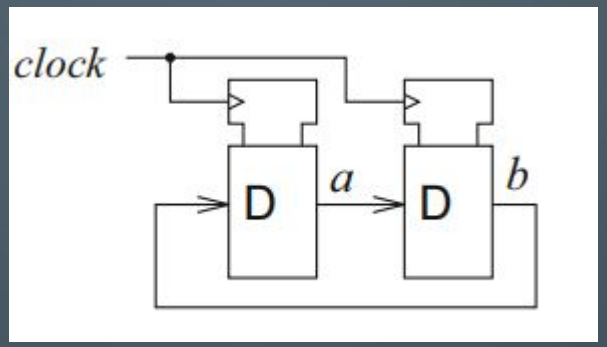
As an example consider the StateMate model of below figure:



- *In the second phase, new values for a and b are stored in temporary variables, say a' and b'. In the final phase, these variables are copied into the user-defined variables:*

phase 2: $a' := b$; $b' := a$;
phase 3: $a := a'$; $b := b'$;

- *As a result, the values of the two variables will be swapped each time an event e happens.*
- *This behavior corresponds to that of two cross-coupled registers (one for each variable) connected to the same clock (see Fig) and reflects the operation of a synchronous (clocked) finite state machine including those two registers*



- *Without the separation into phases, the same value would be assigned to both variables. The result would depend on the sequence in which the assignments were performed.*
- *Due to the separation, the results do not depend on the order in which parts of the model are executed by the simulation.*
- *There are different names for this property: Kahn calls this property determinate. In other papers, this property is called deterministic.*
- *In contrast, non-deterministic finite state machines can be in several states at the same time. Languages may have non-deterministic operators. For these operators, different behaviors are legal implementations. Approximate, non-deterministic computations would be a relevant special case of non-deterministic operators.*

- *Consider Fig. a. If event A takes place while the system is in the left state, we must figure out which transition will take place. If these conflicts would be resolved arbitrarily, then we would have a non-determinate behavior.*

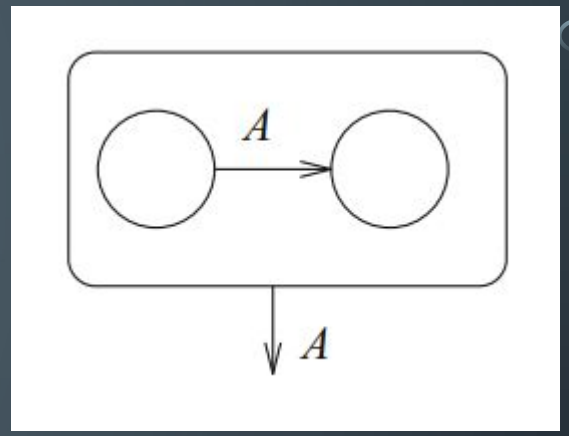


Fig. a

- *Typically, priorities are defined such that this type of a conflict is eliminated. Now, consider Fig. b. There will be a conflict for, e.g., $x = 15$. Such conflicts are difficult to detect.*

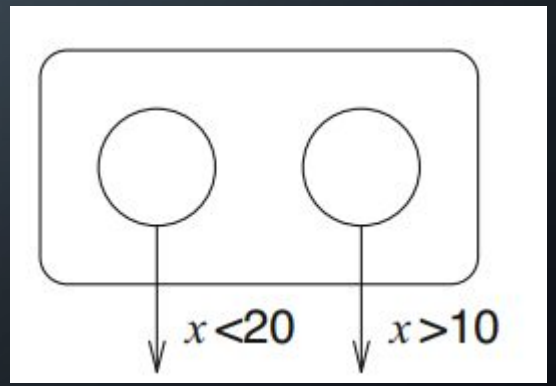


Fig. b

- *Achieving a determinate behavior requires the absence of conflicts that are resolved in an arbitrary manner*

Evaluation And Extensions

- StateMate implicitly assumes a broadcast mechanism for updates on variables. Hence, StateCharts or StateMate can be implemented easily for shared memorybased platforms but are less appropriate for message passing and distributed systems.
- These languages essentially assume shared memory-based communication, even though this is not explicitly stated.
- For distributed systems, it will be very difficult to update all variables between two steps. Due to this broadcast mechanism, StateMate is not an appropriate language for modeling distributed systems. Hence, StateCharts' main application domain is that of local, control-dominated systems.
- The capability of nesting hierarchies at arbitrary levels, with a free choice of AND-and OR-superstates, is a key advantage of StateCharts.

Evaluation And Extensions

- Another advantage is that the semantics of StateMate is defined at a sufficient level of detail.
- Furthermore, there are quite a number of commercial tools based on StateCharts. StateMate and StateFlow are examples of commercial tools based on StateCharts.
- Many of them are capable of translating StateCharts into equivalent descriptions in C or VHDL.
- From VHDL, hardware can be generated using synthesis tools. Therefore, StateCharts-based tools provide a complete path from StateCharts-based specifications down to hardware.
- Generated C programs can be compiled and executed. Hence, a path to software-based

- *Unfortunately, the efficiency of the automatic translation is sometimes a concern.*
- *For example, we could map substates of AND-superstates to processes at the operating system level. This would hardly lead to efficient implementations on small processors.*
- *The productivity gain from object-oriented programming is not available in StateCharts, since it is not object-oriented.*
- *StateCharts do not comprise program constructs for describing complex computation and cannot describe hardware structures or non-functional behavior.*

- *State Charts allows timeouts.*
- *There is no straightforward way of specifying other timing requirements.*
- *UML includes a variation of StateCharts and hence allows modeling state machines. In UML, these diagrams are called state diagrams in version 1 of UML and state machine diagrams from version 2.0*