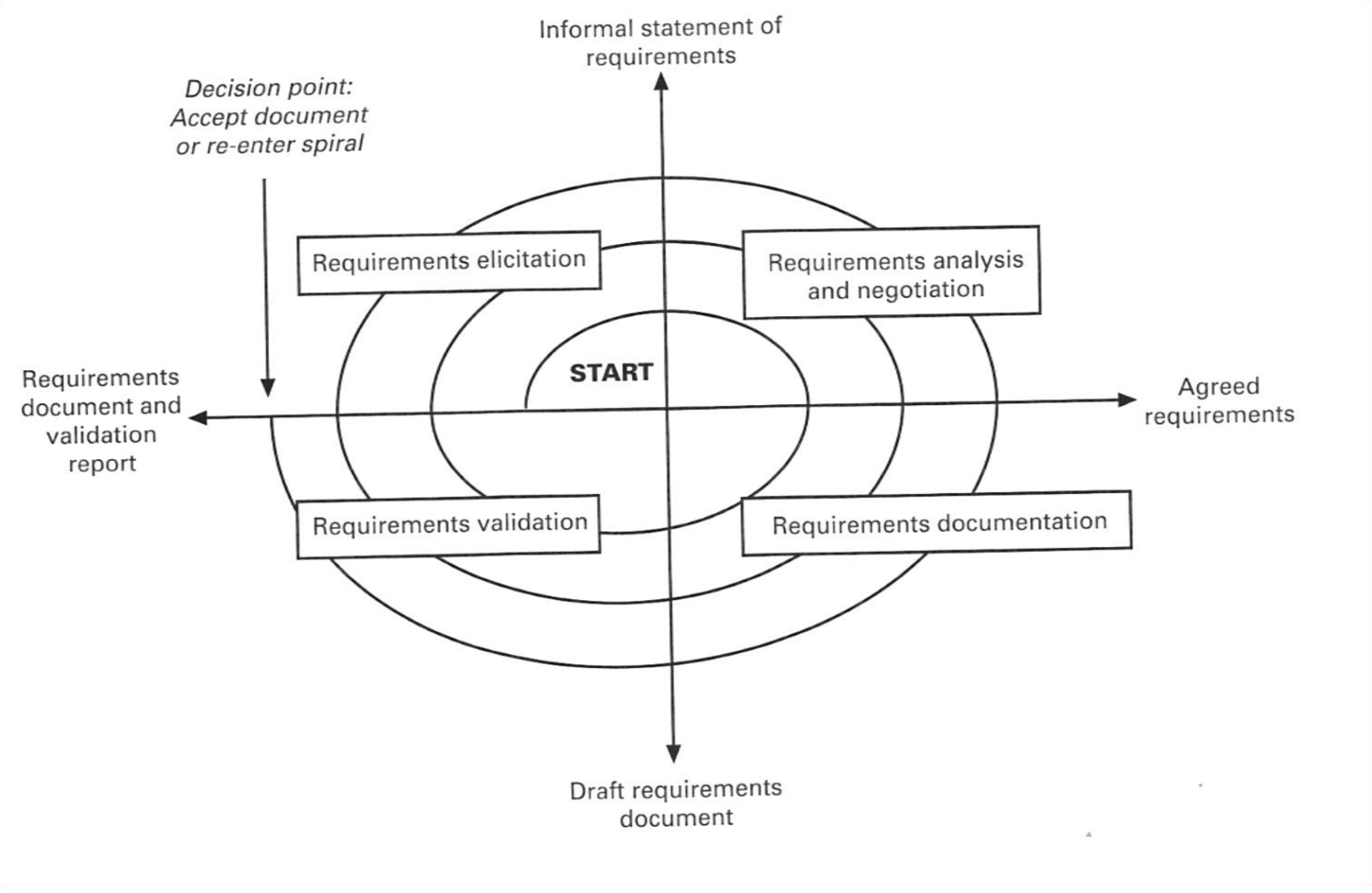
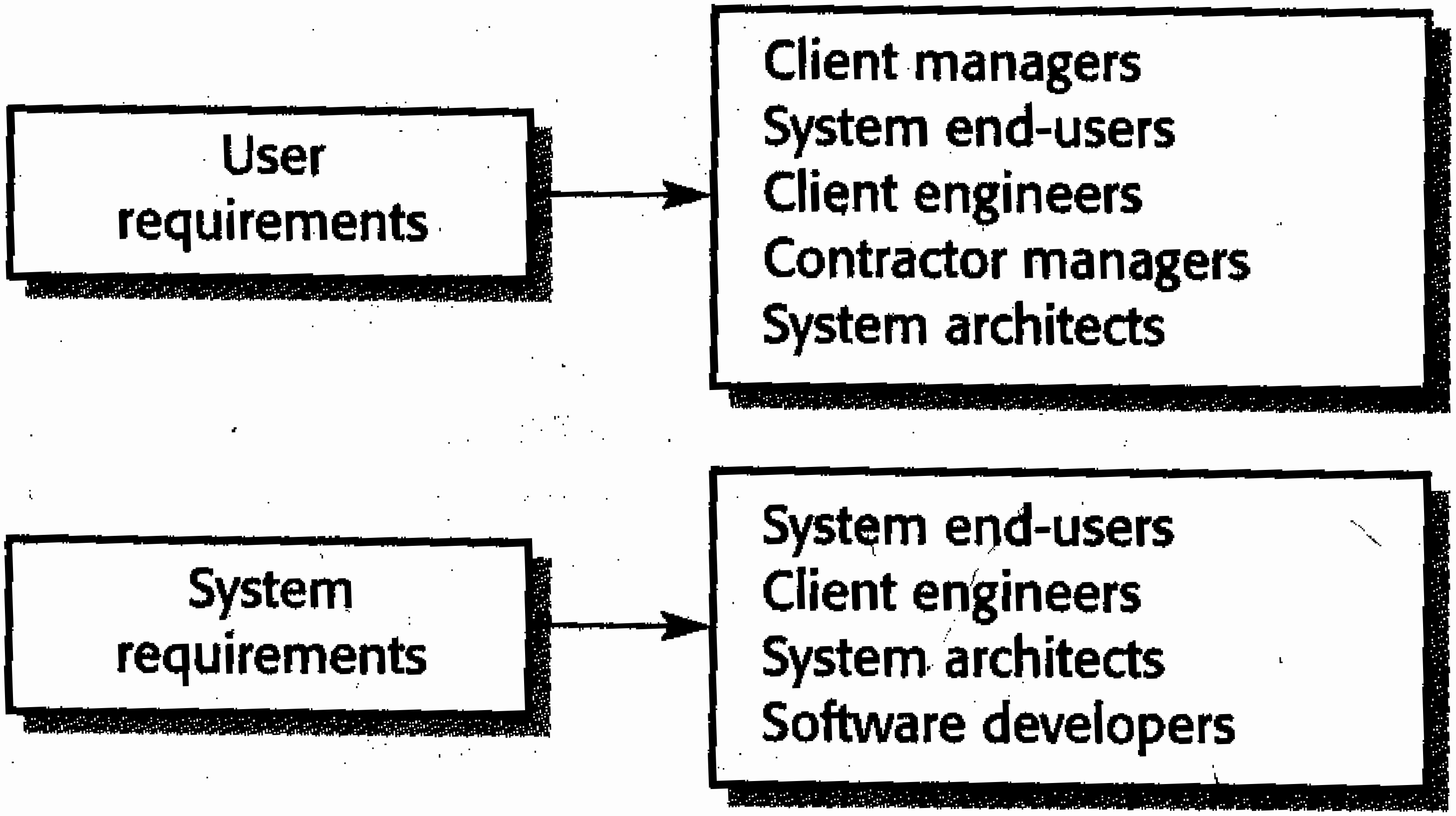
Software Requirements Engineering: a brief review

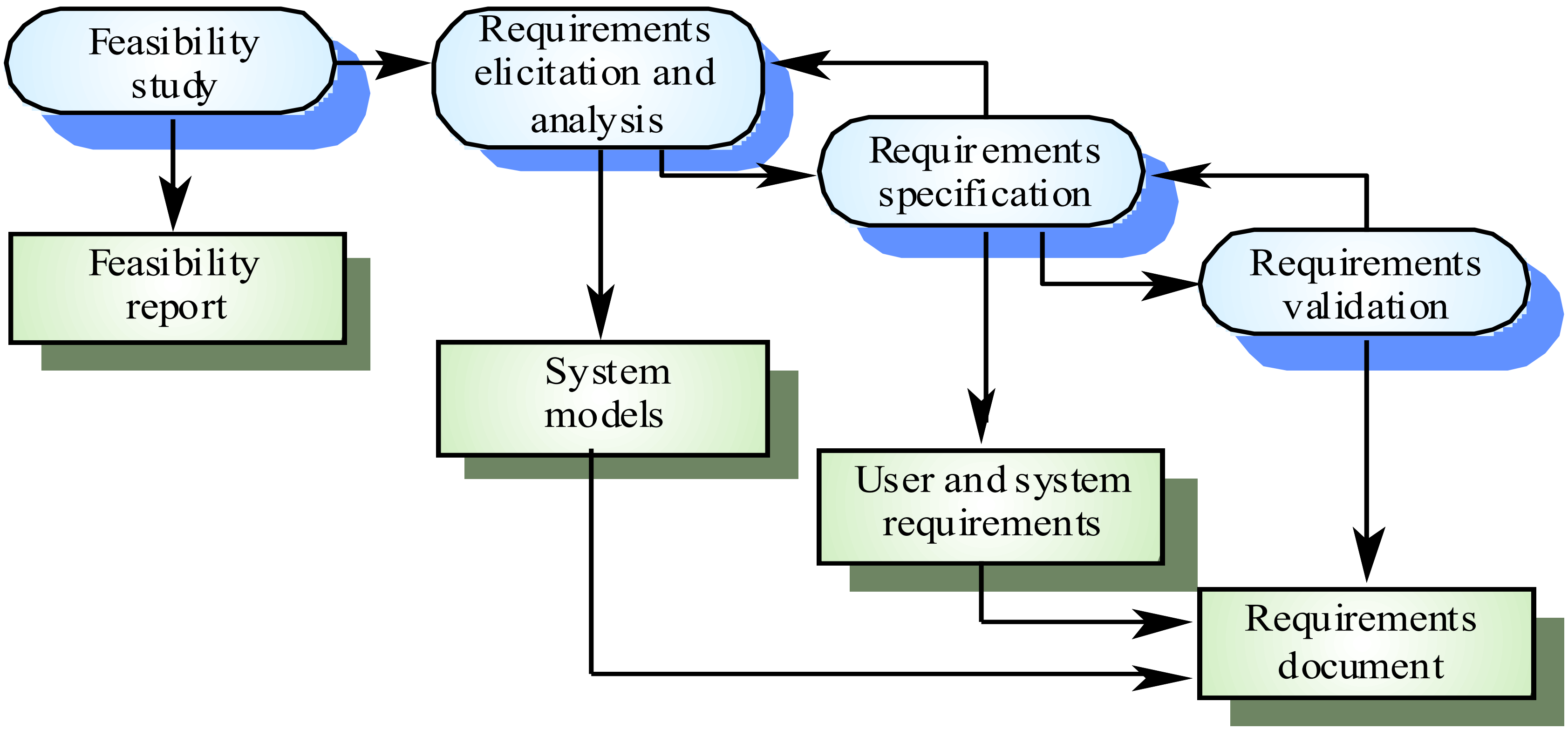
Software Requirements Engineering: a spiral model



User and System requirements: a general view



Requirements engineering: process



Specification as a process

Process of identifying stakeholders, eliciting requirements, compiling them…

Completeness

The specification must capture *all* the relevant requirements of the product

Minimality

The specification must capture *nothing but* the relevant requirements of the product

Specification as a product (1)

Formality

Specification must be represented in such a way as to describe *precisely* what functional behavior is required

Abstraction

Specification must describe what requirements the software product must satisfy, not how to satisfy (*what* rather than *how*)

Specification as a product (2)

Precision

should state *exactly* what is desired of the system

Completeness

include descriptions of *all* facilities required

Consistency

there should be no conflicts or contradictions in the descriptions of the system facilities

Software Requirements Specification (SRS) document

Table of Contents

---

* The reference for Table 4.1 is IEEE Std. 830-1993, "Recommended Practice for Software Requirements Specifications," *Standards Collection on Software Engineering*, IEEE Press, New York, 1994.

System Requirements

Functional requirements

•Statements of services the system should provide
•How the system should react to particular inputs
•How the system should behave in particular situations
•State what the system should not do

Non-functional requirements

Requirements that are not directly concerned with the specific functions delivered by the system. They may relate to properties such as reliability, security, performance, availability, response time, capabilities of I/O devices, etc.

Functional Requirements

Completeness

all services required by the user are defined

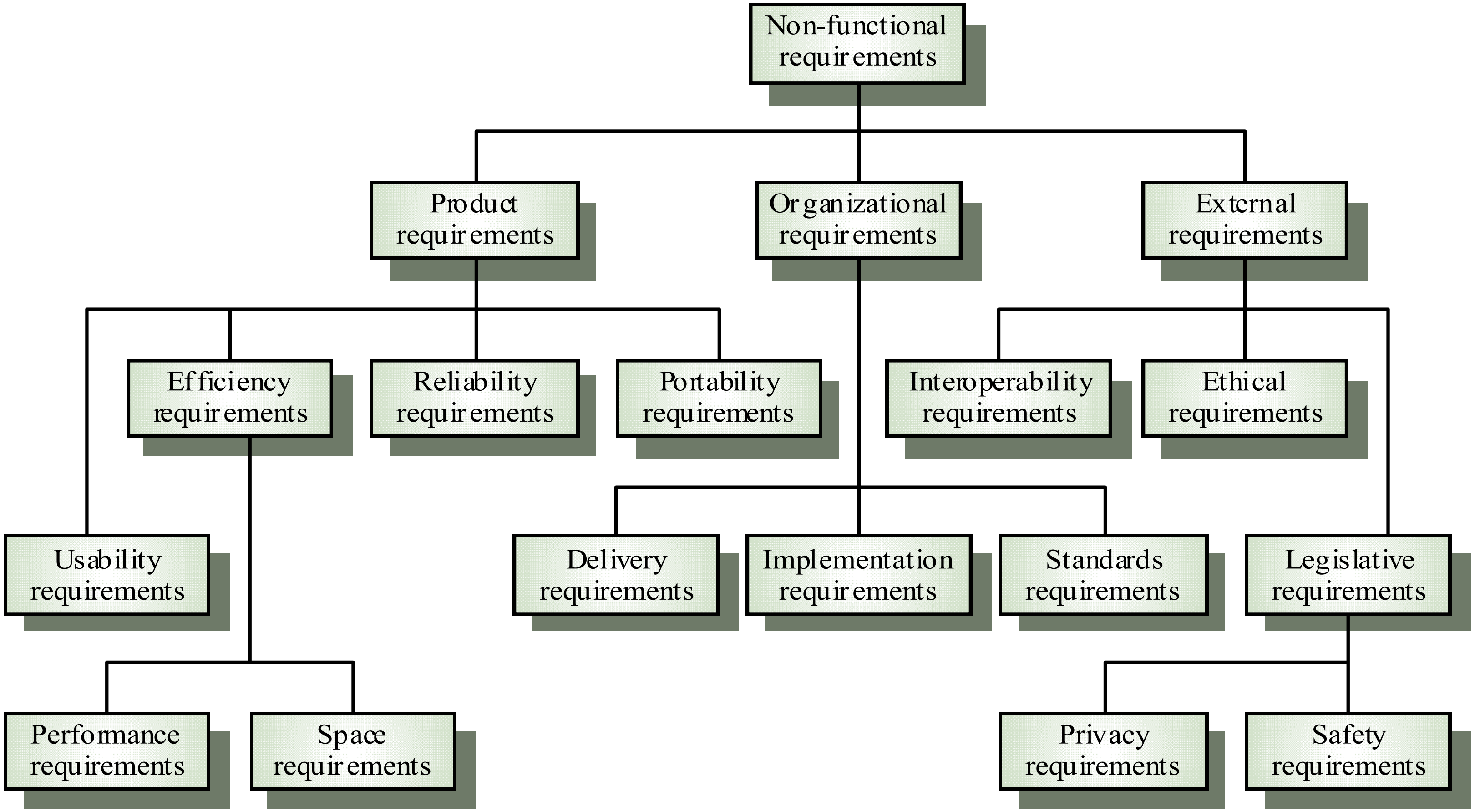
Consistency

There should not be contradictory definitions

Mistakes and omissions– why?

- **Complexity and size of system**
- **Different system stakeholders – different and often inconsistent needs**

Non-functional requirements



Non-functional Requirements: metrics

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target-dependent statements Number of target systems

User requirements

Should describe functional and non-functional requirements so that they are understandable by system users without detailed technical knowledge

Simple language, tables and forms, intuitive diagrams

Given these, various problems could emerge

•**Lack of clarity**
•**Requirements confusion-**- requirements may not be clearly distinguished
•**Requirements amalgamation**– several different requirements may be expressed together as a single requirement

Specification styles

Two independent, orthogonal criteria:

Formal and ***informal*** specifications

Operational and ***descriptive*** specifications

Informal specifications – natural language
Formal specifications – precise syntax and semantics

Operational specifications: describe the desired behavior

Descriptive specifications: describe desired properties in a declarative fashion

Specification styles: sorting an array

Informal and operational specification

Let `a` be an array of `n` elements. The result of sorting `a` is an array `b` of `n` elements that may be built as follows:

1. Find the smallest element in `a`, and assign it as the first element in `b`
2. Remove the element you found in step 1 from `a`, and find the smallest of the remaining elements. Assign the element as the second element in `b`
3. Repeat steps 1-2 until all elements have been removed from `a`

Descriptive specification

The result of sorting `a` is an array that is a permutation of `a` and is sorted

Categories of operational specifications

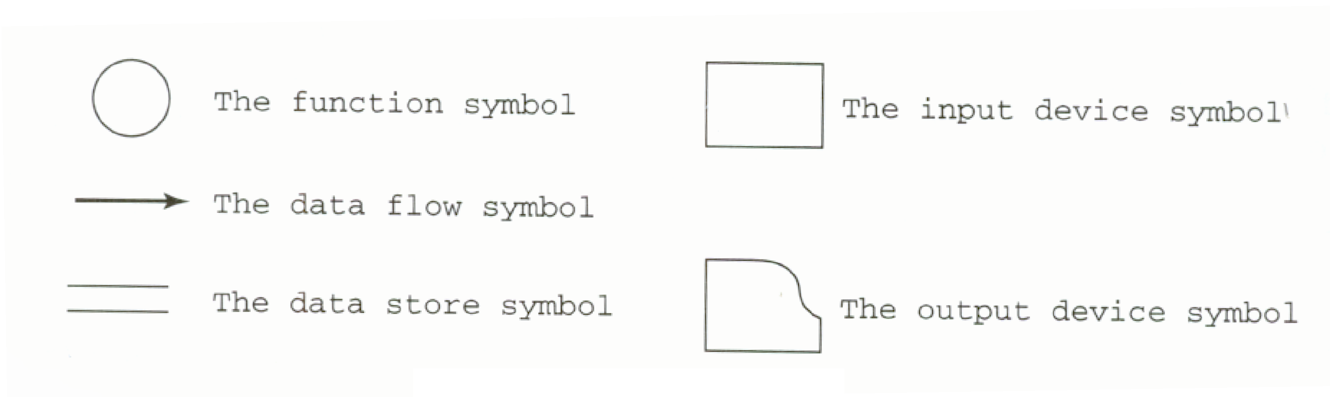
Data flow diagrams (DFDs)

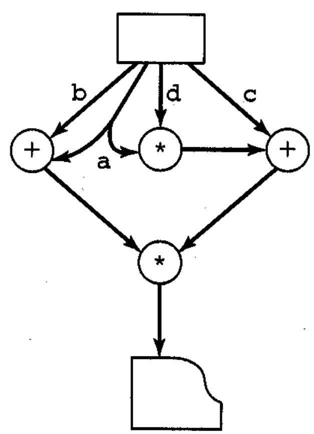
Finite State Machines (FSMs) and their variants

Petri nets (PNs) and their variants

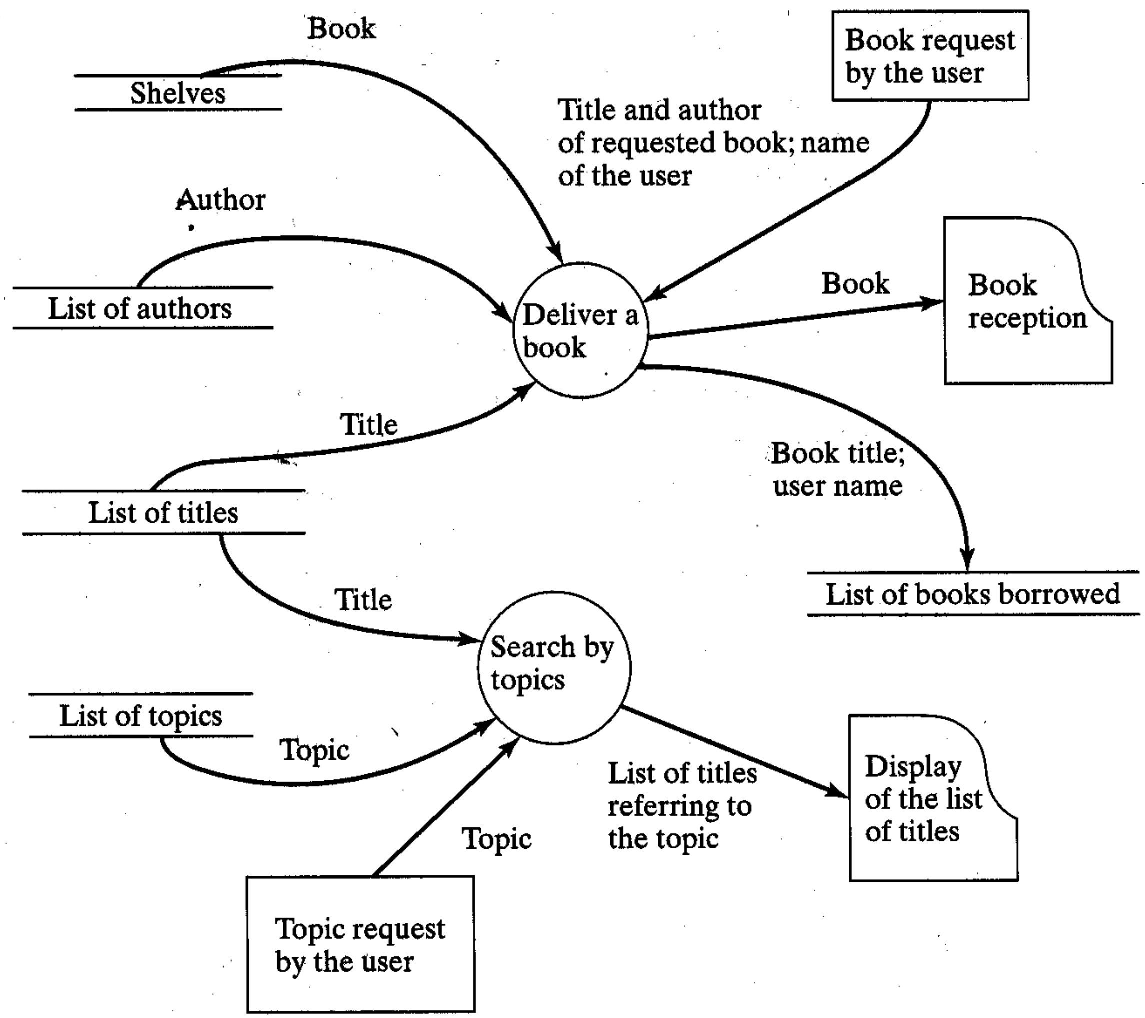
Data flow diagrams

semiformal notation focused on flow of data

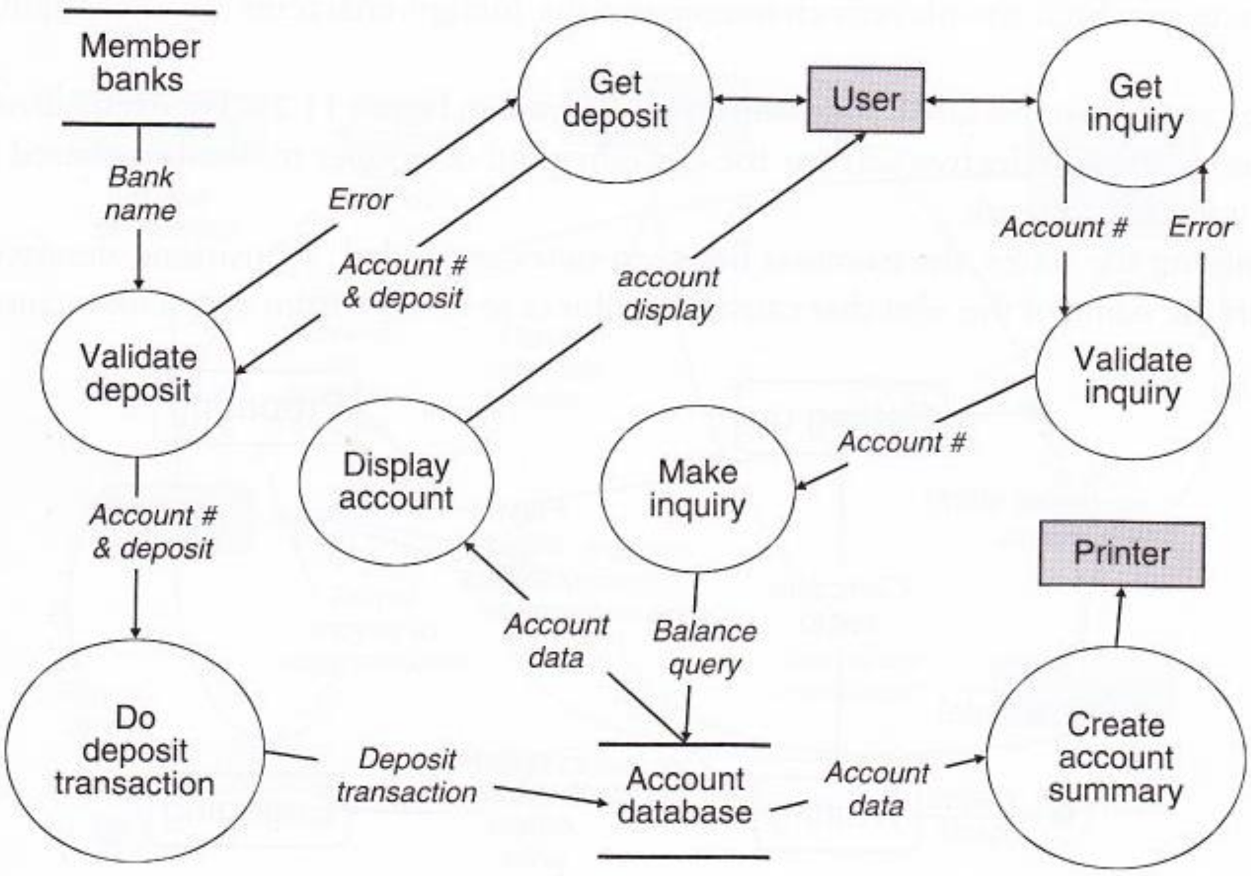




Data flow diagrams – example 1



Data flow diagrams – example 2



Finite State Machines

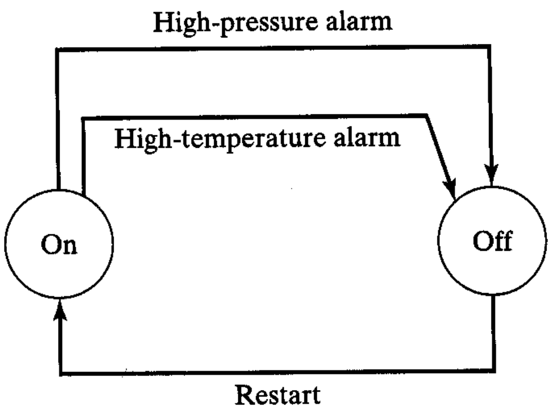
Directed graph

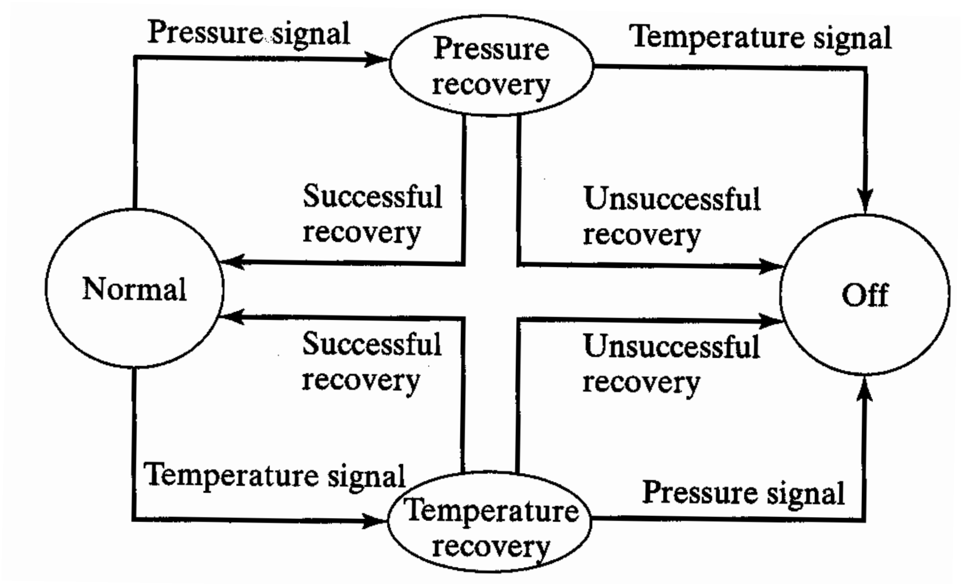
Nodes – states
Edges – transitions (actions)

Two main types of finite state machines:

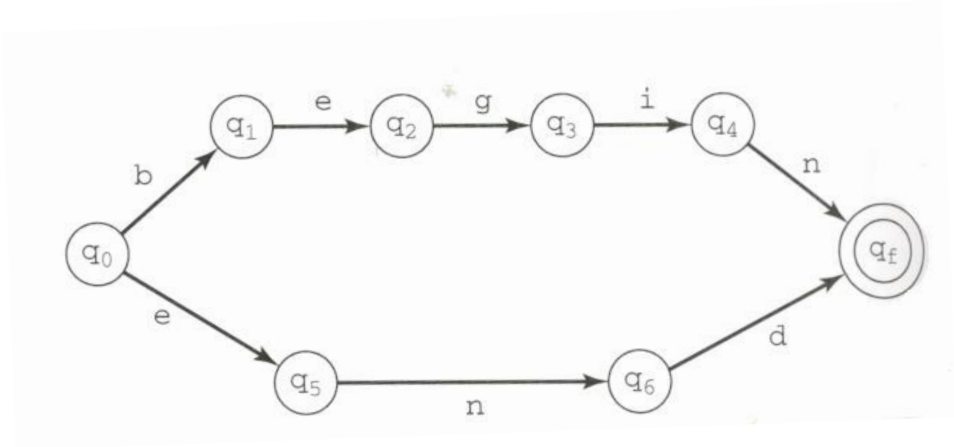
MOORE: output = f(state)
MEALY: output =f (state, input)

Finite State Machines

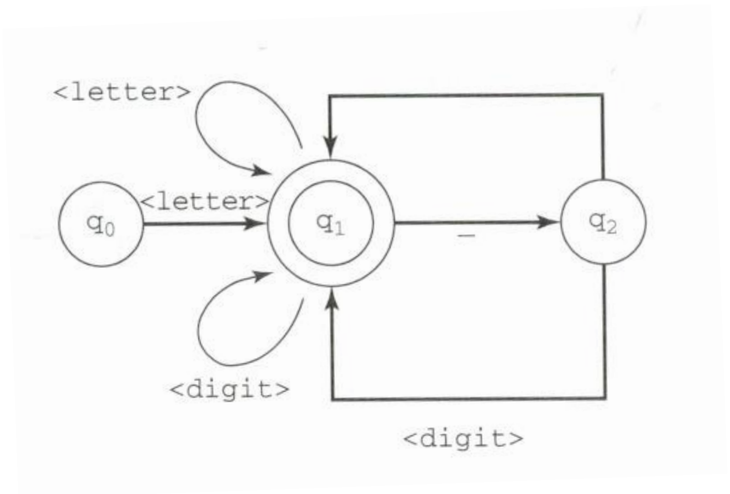




Finite State Machines

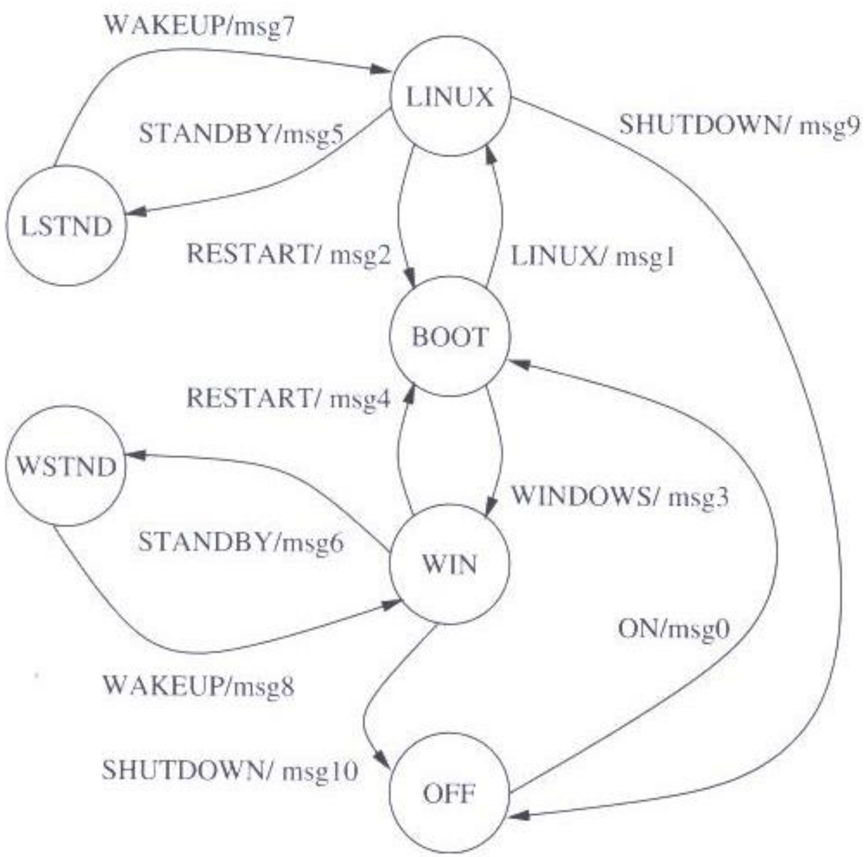


Acceptance of keywords
begin
end



Acceptance of identifiers

Finite State Machine: dual boot laptop

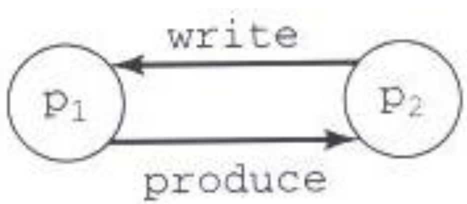


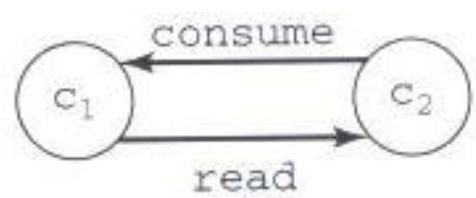
LSTNB- standby mode for the Linux mode
WSTNB- standby mode for the Windows mode
OFF state – also by using the power button; not shown here

Producer-consumer system

Producer produces messages and places them in a two-slot buffer.
A consumer reads the messages and removes them from the buffer.
If the buffer is full, the producer must wait until the consumer has emptied the slot.
If the buffer is empty, the consumer process must wait until the producer has Inserted a message.

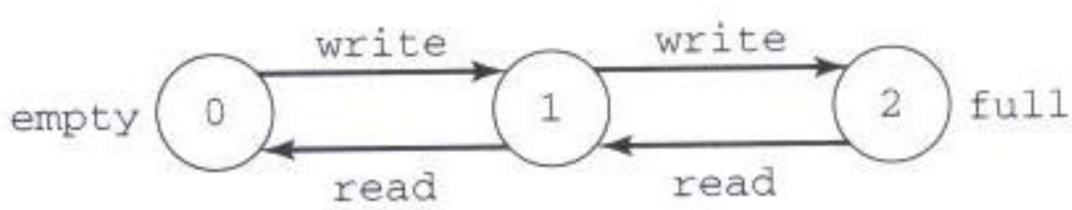
producer





consumer

buffer



Producer-consumer system

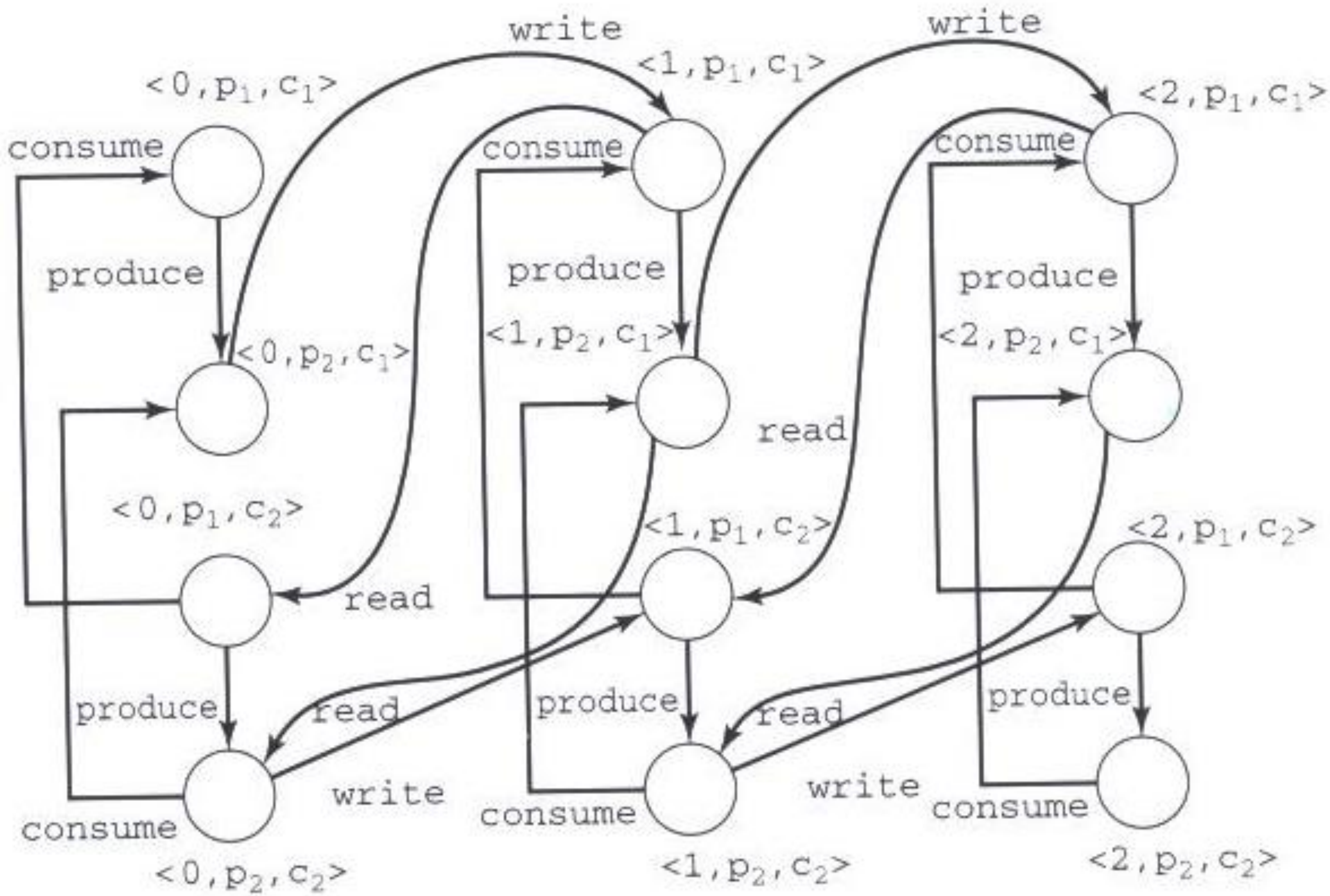
Composition of finite state machines:

States – Cartesian product of the component state sets

< 0, p, c>

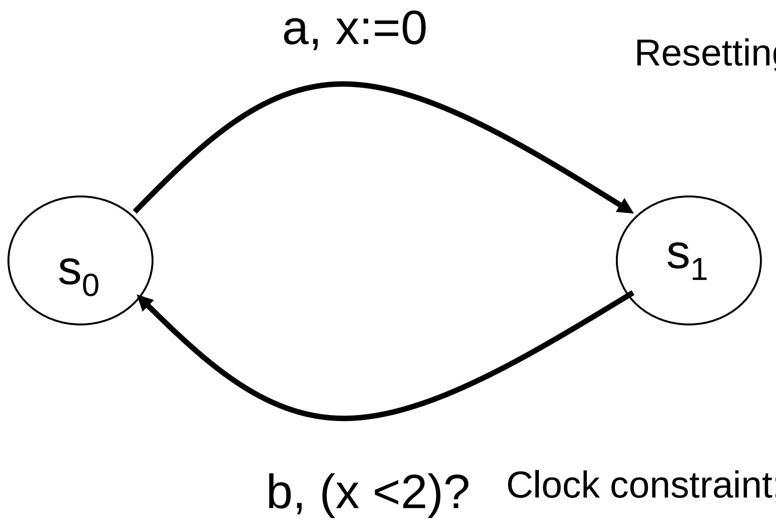
State of buffer   state of producer   state of consumer

Producer-consumer system



Timed automata

Incorporation time with inputs (symbols); association of clocks with transitions

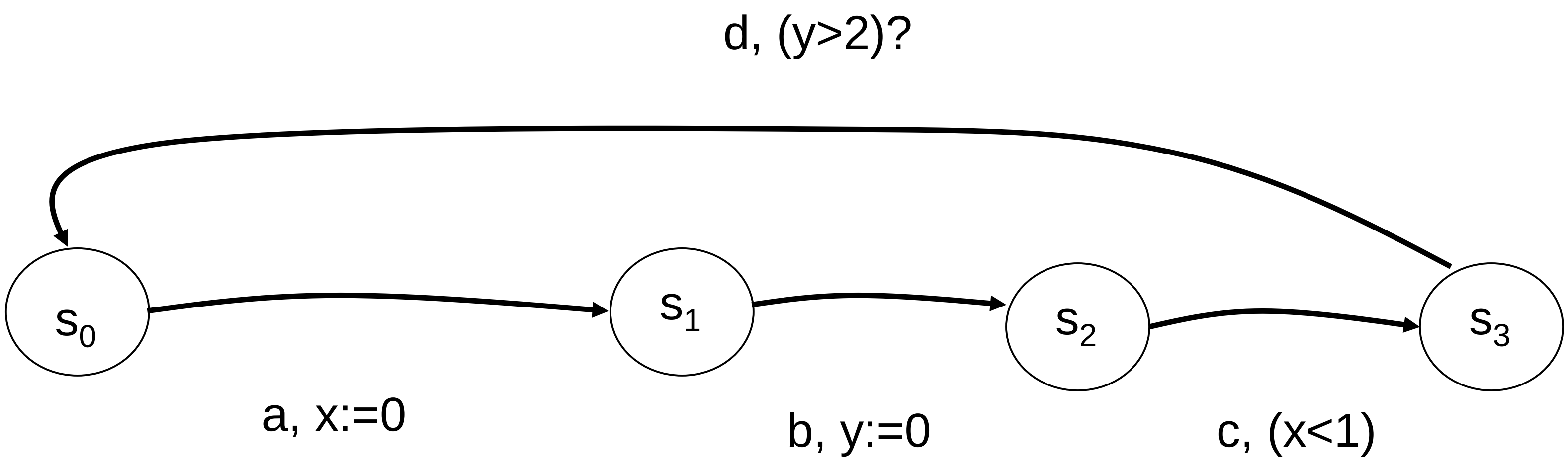


Resetting the clock x when edge traversed

Clock constraint; transition enabled if x less than constraint

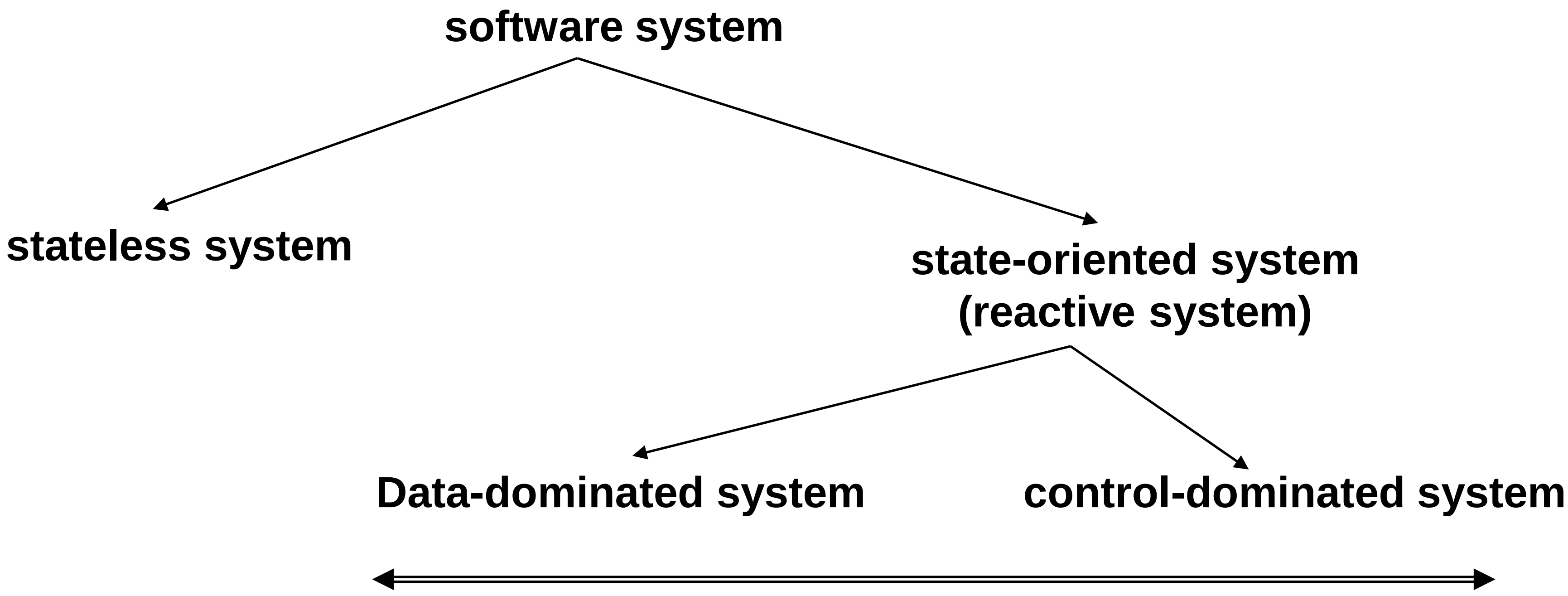
Timed automata

Use of 2 clocks

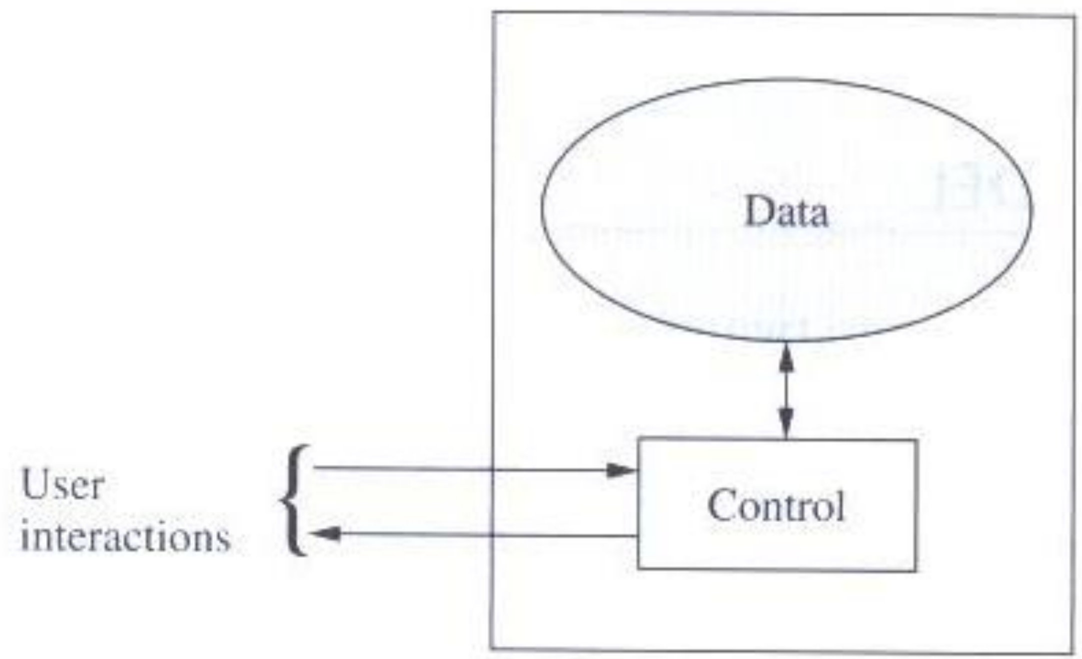


Delay between “b” and “d” greater than 2

Software – a taxonomy



Data-dominated systems

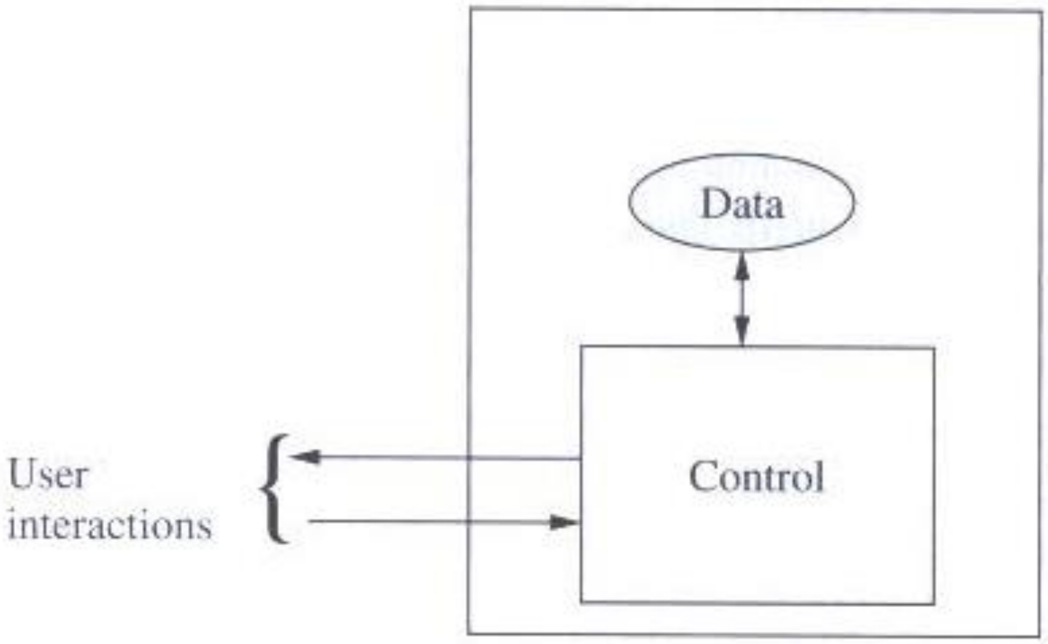


Web browsing application: accessing remote data, formatting for display

not too much state info; BACK info

not a time-dependent application (still depends on the TCP/IP operations)

Control-dominated systems



Complex interactions with the user; relatively small amount of data processed.
Telephone switching system

Petri nets:

Specification of systems that involve *parallel* or *concurrent* activities

Formal definition:

$$(P, T, F)$$

Where

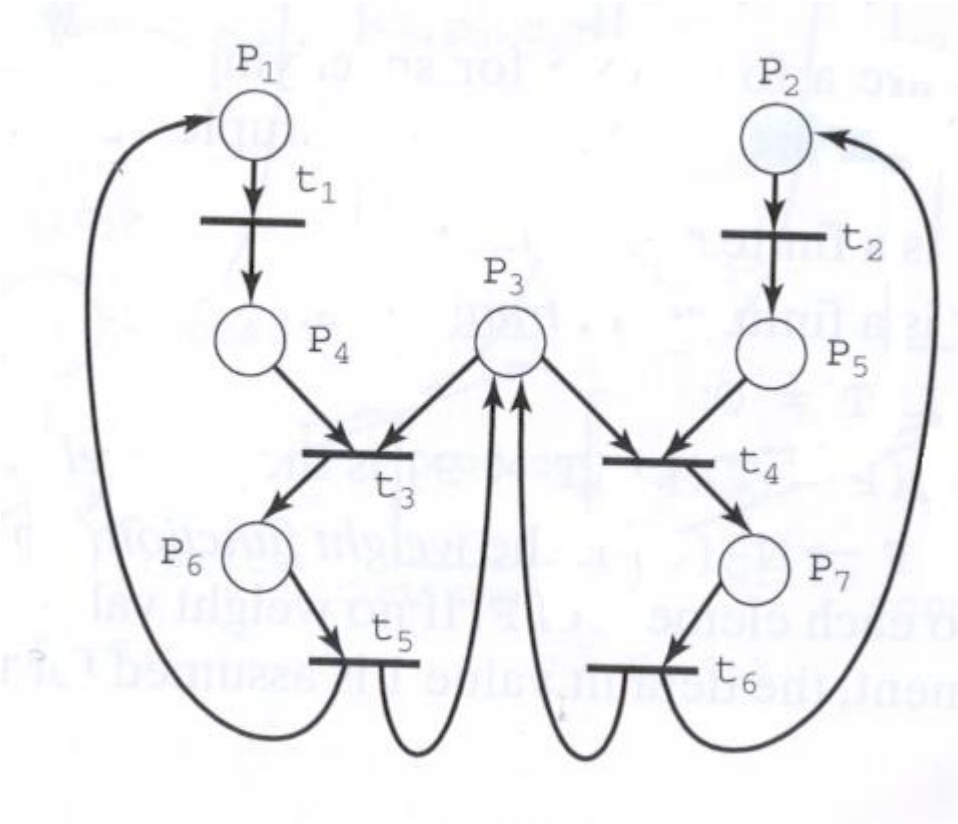
P- a finite set of places
T- a finite set of transitions
F – flow relation defined over $(P \times T) \cup (T \times P)$

Marking M

$$M: P \rightarrow \mathbf{N}$$

N – natural numbers

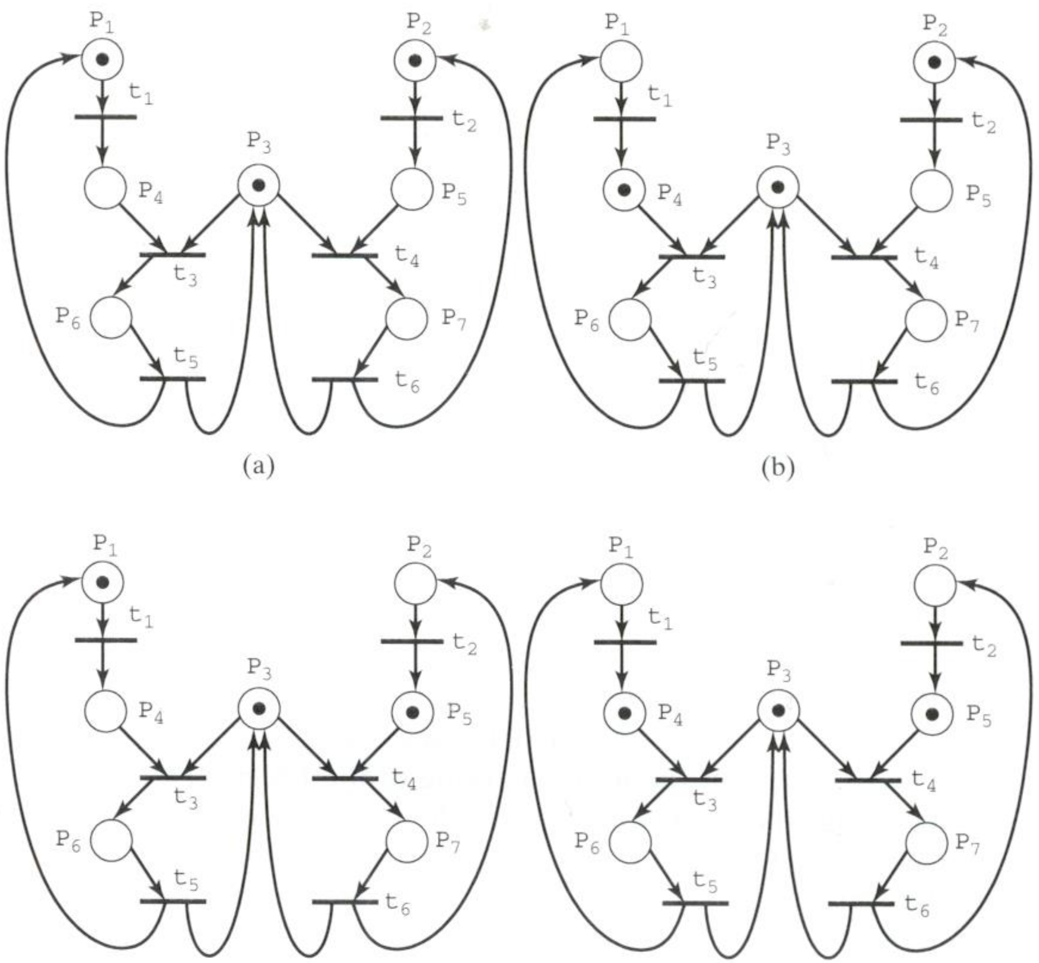
Petri nets: example



Petri nets: interpretations

Input places	**transition**	**output places**
precondition	event	postcondition
input data	computing	output data
resources needed	task/job	resources released
buffer	processor	buffer

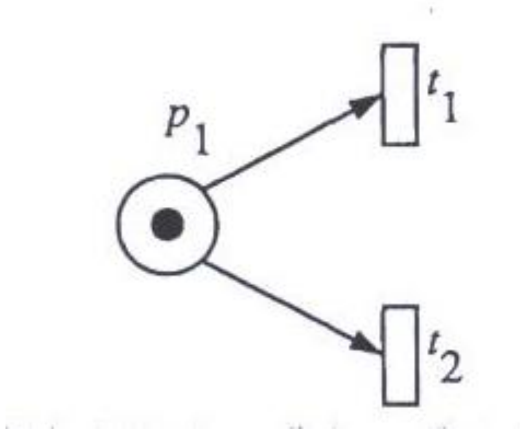
Petri nets: specifying asynchronous systems

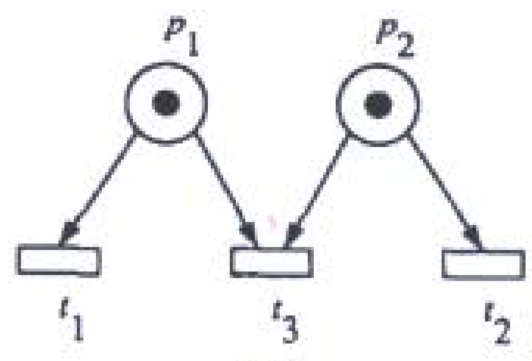


(a) (b)

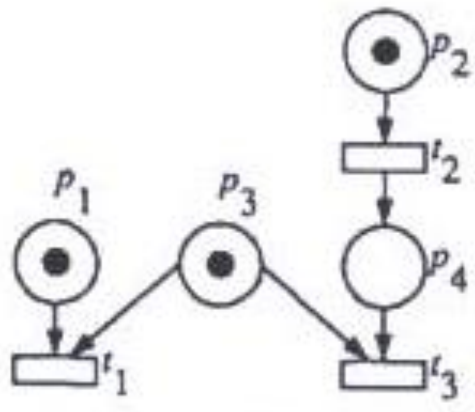
(c) (d)

Petri nets: modeling conflict, choice, decision, nondeterminism



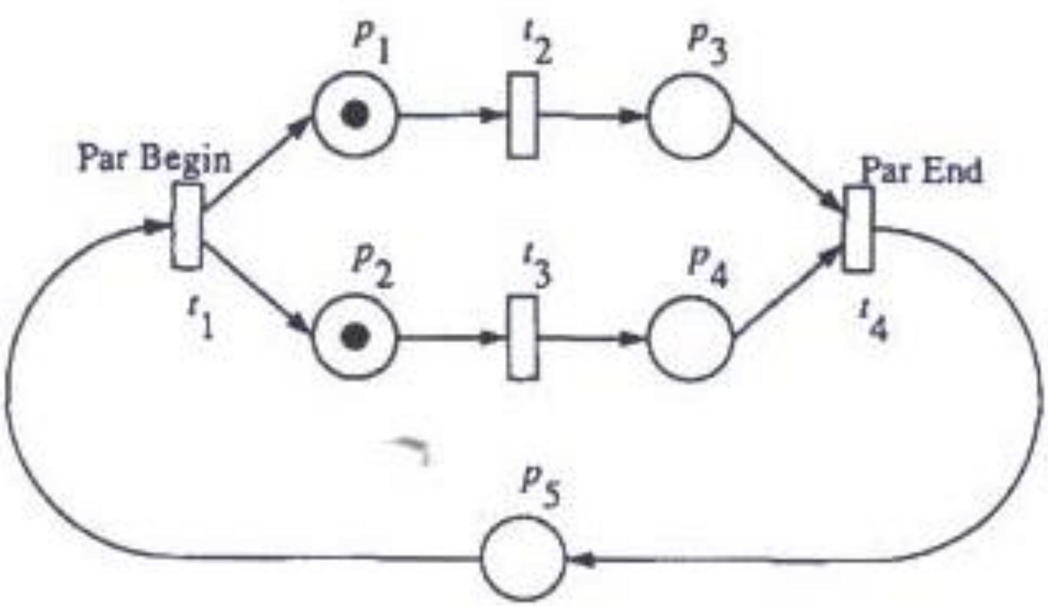


t1 and t2 are concurrent but in conflict with t3



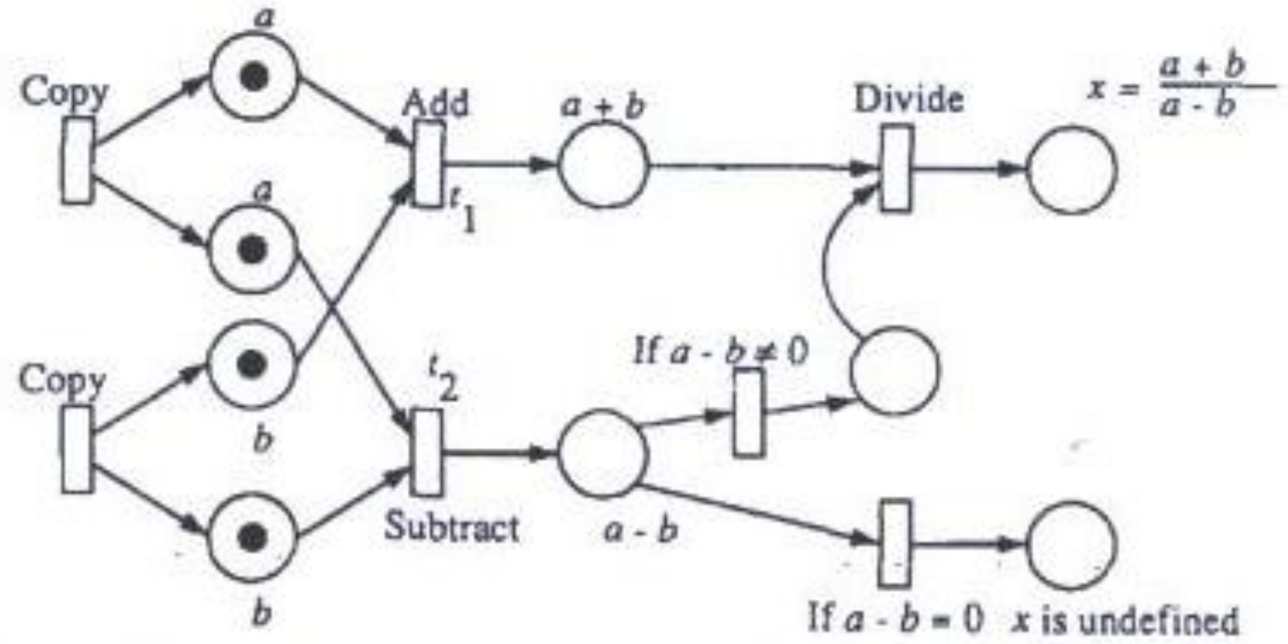
t1 is concurrent with t2 but will be in conflict with t3 if t2 fires before t1

Deterministic parallel activities

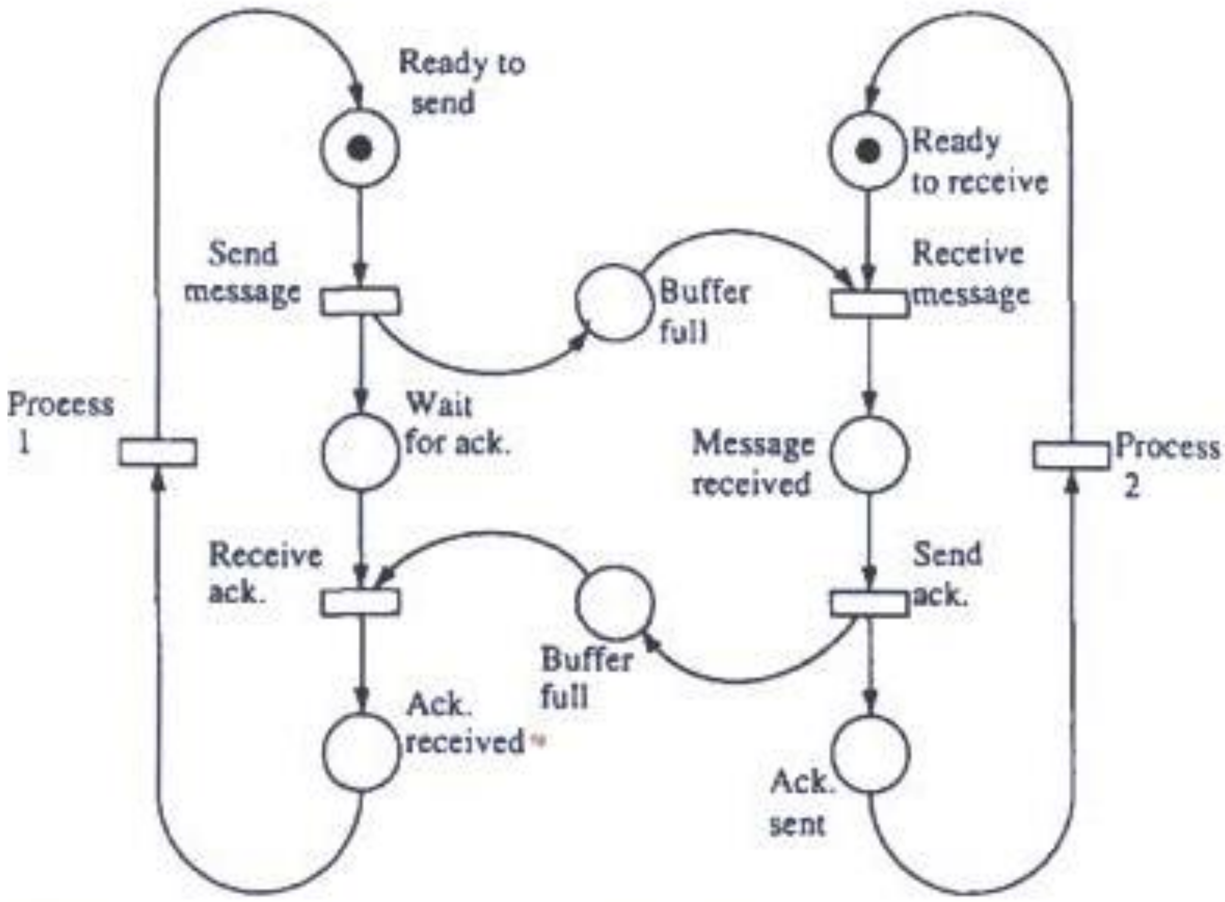


Marked graph (each place has a single incoming arc and outgoing arc)
Representation of **<u>concurrency</u>**

Dataflow modeling

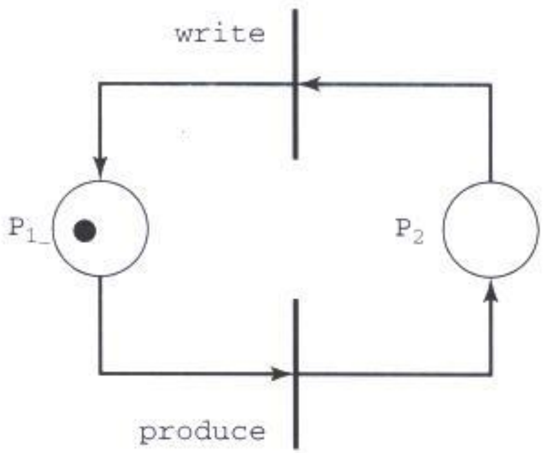


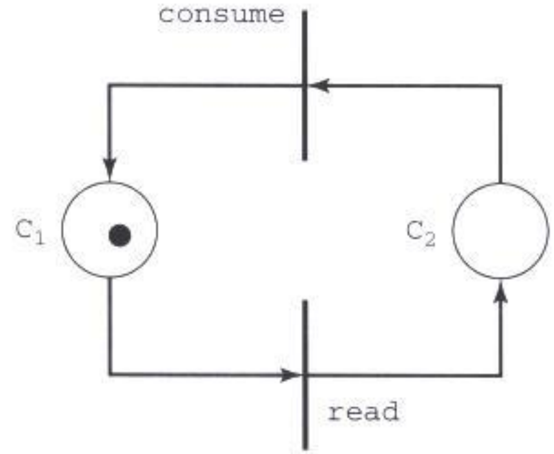
Communication protocol



Petri nets: producer-consumer (1)

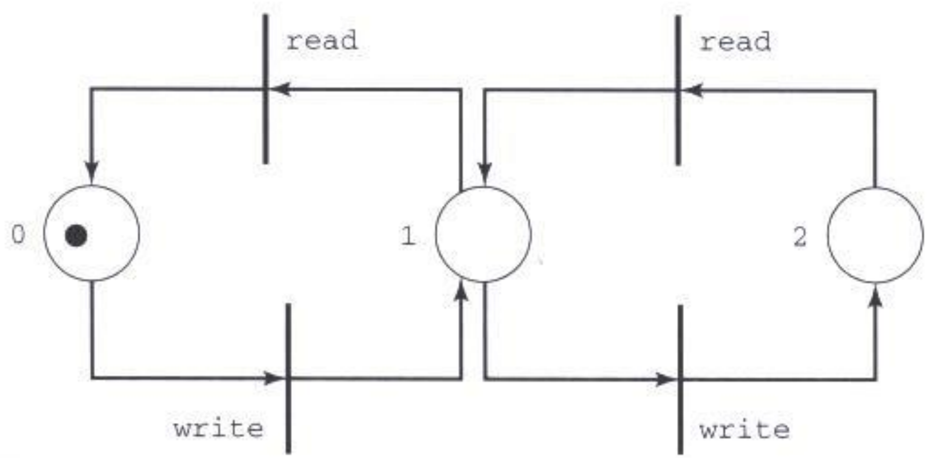
producer





consumer

buffer

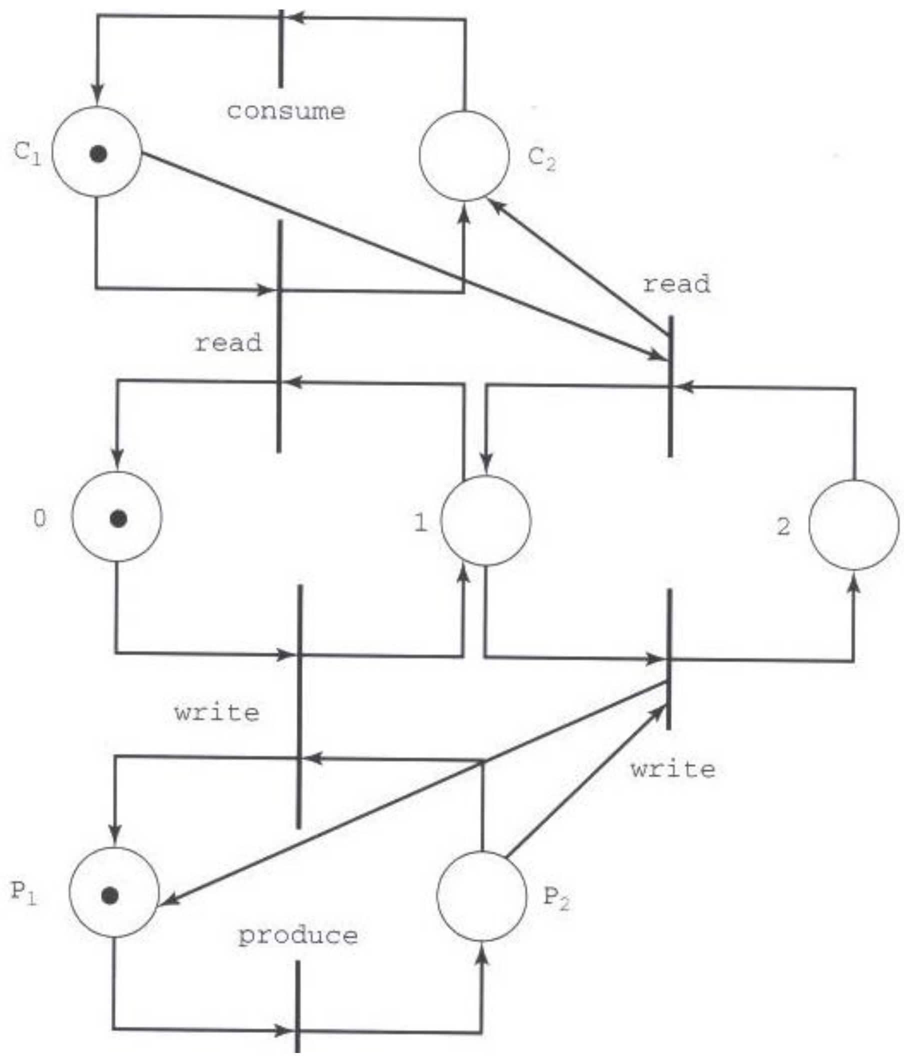


Petri nets: producer-consumer (2)

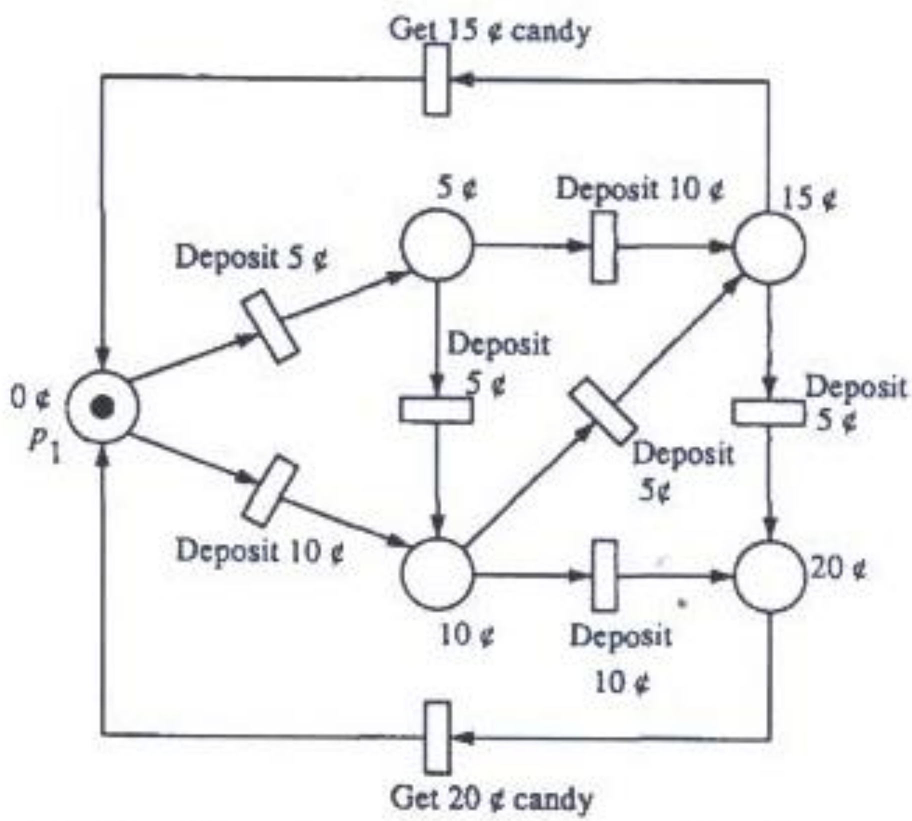
consumer

buffer

producer



Petri nets and finite state machines



each transition: *single* input place and *single* output place

Descriptive specifications

Describe *desired properties* of a system (rather than a *desired behavior*)

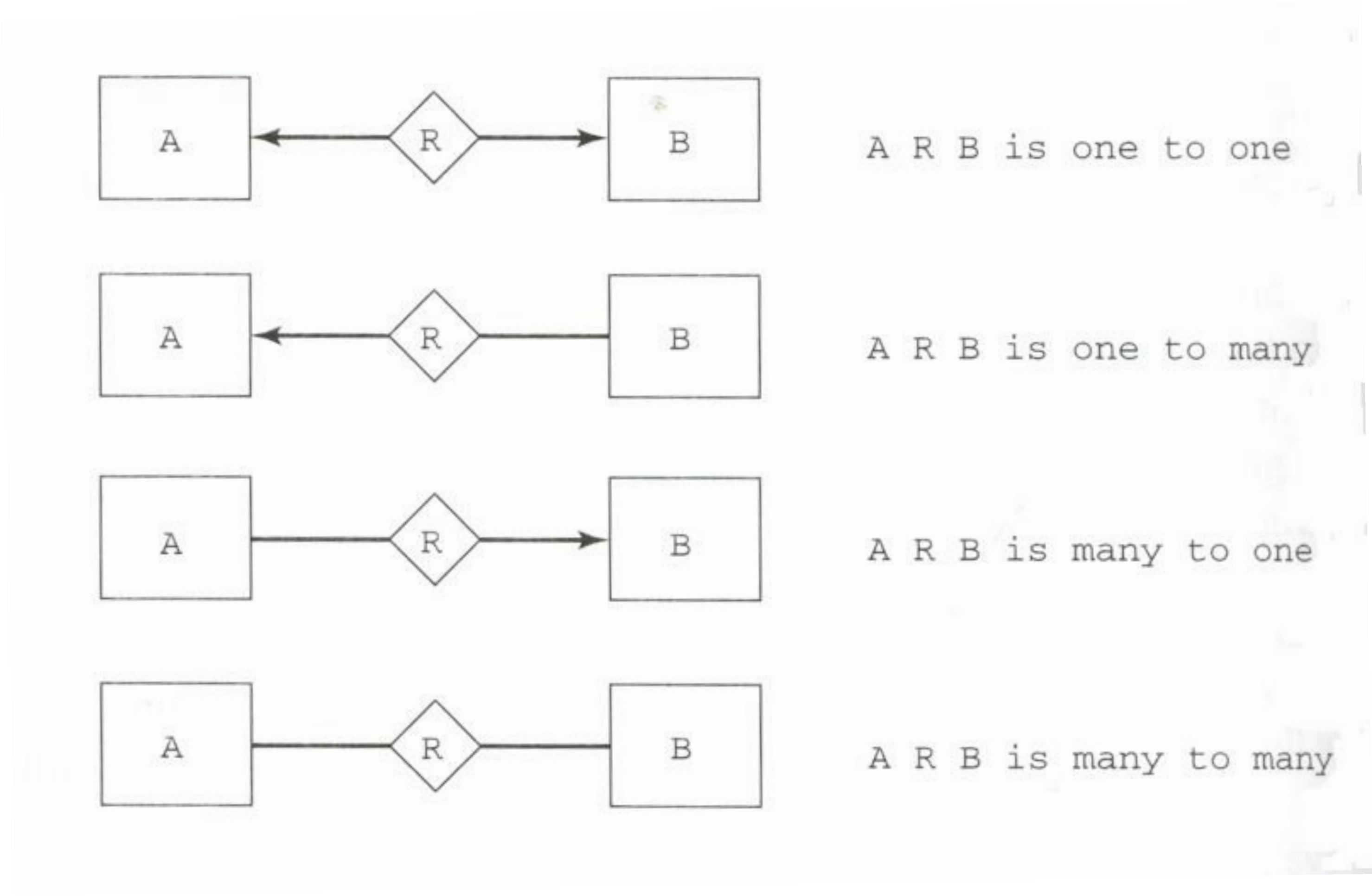
Typically mathematical formulas

Entity –Relationship Diagrams (ER Diagrams)

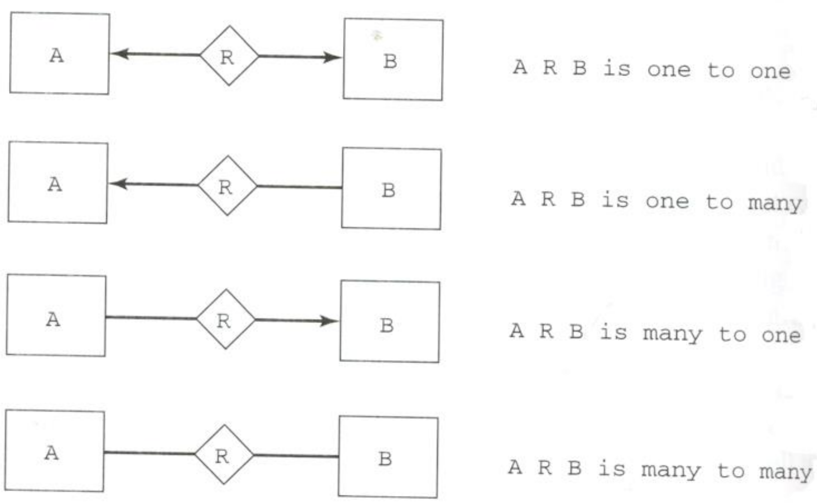
Logic specifications (e.g., Z notation)

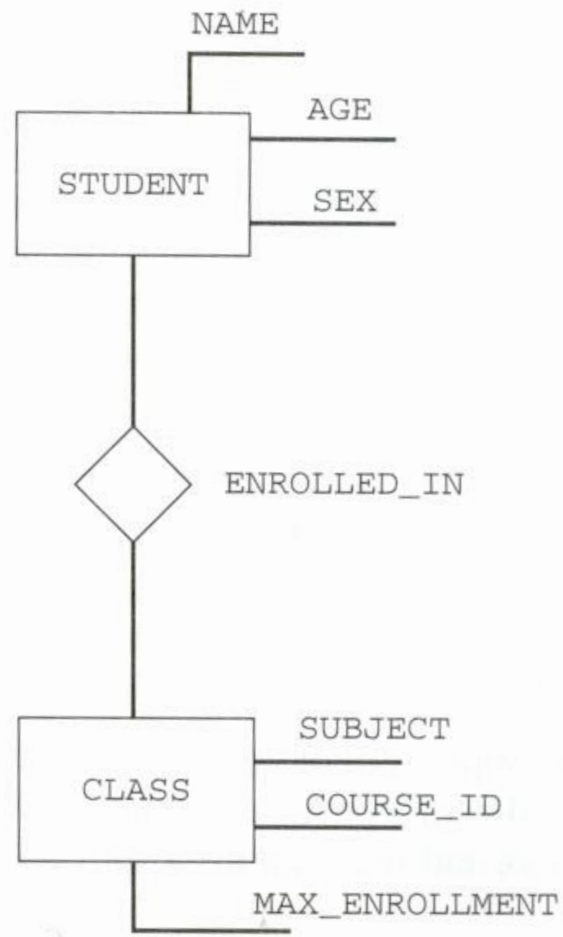
Algebraic specifications

Entity-relationship diagrams



Entity-relationship diagrams





Logic specifications

First-order theory (FOT)

Formula of FOT: expression involving variables, numeric constants, functions, predicates, brackets, logic connectives – `and, or, not, implies` **quantifiers –** `exists, for all`

```
x > y and y > z implies x > z;
x = y ≡ y = x;
for all x, y, z (x > y and y > z implies x > z);
x + 1 < x - 1;
for all x (exists y(y = x + z));
x > 3 or x < -6.
```

Specifying sequential program `P`

Logic specifications: specifying sequential program

sequential program P

{Pre $(i_1, i_2, \ldots, i_n)$} ← Precondition of P (FOT)

P

{post$(o_1, o_2, \ldots, o_m, i_1, i_2, \ldots, i_n)$} ← Postcondition of P (FOT)

Design by contract

assert- statement, if- statement
support – e.g., Eiffel language
relationship with black box testing

Example (1)

```
int compute_r(int x, int y)
    {
    int r = x;
    int q = 0;
    while  (r > y)
        {
          r = r - y;
          q = q + 1;
        }
    return r;
  }
```

assertion x = y*q + r

Example (2)

assertion y>0

```
int compute_r(int x, int y)
        {
    int r = x;
    int q = 0;
    while  (r > y)
         {
           r = r - y;
           q = q + 1;
         }
    return r;
   }
```

assertion x = y*q + r

Example(3)

precondition y>0

```
int compute_r(int x, int y)
{
  int r = x;
  int q = 0;
  while  (r > y || r == y)
     {
       r = r - y;
       q = q + 1;
     }
  return r;
}
```

postconditions x = y*q + r and r < y