Specification

Outline

- Discussion of the term "specification"
- Types of specifications
  - operational
    - Data Flow Diagrams
    - (Some) UML diagrams
    - Finite State Machines
    - Petri Nets
  - descriptive
    - Entity Relationship Diagrams
    - Logic-based notations
    - Algebraic notations

Specification

- A broad term that means *definition*
- Used at different stages of software development for different purposes
- Generally, a statement of agreement (*contract*) between
  - producer and consumer of a service
  - implementer and user
- All desirable qualities must be specified

Uses of specification

- Statement of user requirements
  – major failures occur because of misunderstandings between the producer and the user
  – "The hardest single part of building a software system is deciding precisely what to build" (F. Brooks)

Uses of specification (cont.)

- Statement of the interface between the machine and the controlled environment
  - serious undesirable effects can result due to misunderstandings between software engineers and domain experts about the phenomena affecting the control function to be implemented by software

Uses of specification (cont.)

- Statement of requirements for implementation
  - design process is a chain of specification (i.e., definition)–implementation–verification steps

Specification qualities

- Precise, clear, unambiguous
- Consistent
- Complete
  - internal completeness
  - external completeness
- Incremental

Clear, unambiguous, understandable

- Example: specification fragment for a word-processor

```
Selecting is the process of designating
areas of the document that you want to
work on. Most editing and formatting
actions require two steps: first you
select what you want to work on,
such as text or graphics; then you
initiate the appropriate action.
```

can an area be scattered?

Precise, unambiguous, clear

- Another example (from a real safety-critical system)

```
The message must be triplicated. The three
copies must be forwarded through three
different physical channels. The receiver
accepts the message on the basis of a
two-out-of-three voting policy.
```

can a message be accepted as soon as we receive 2 out of 3 identical copies of message or do we need to wait for receipt of the 3rd?

Consistent

- Example: specification fragment for a word-processor

```
The whole text should be kept in lines
of equal  length. The length is specified
by the user. Unless the user gives an
explicit hyphenation command,
a carriage return should occur only
at the end of a word.
```

What if the length of a word exceeds the length of the line?

Complete

- Internal completeness
- External completeness

Incrementality principle

- Due to difficulty in achieving complete, precise and unambiguous specifications.
- Referring to the specification process
  - start from a sketchy document and progressively add details
- Referring to the specification document
  - document is structured and can be understood in increments

Classification of specification styles

- Informal, semi-formal, formal
  - Informal: written in natural language , uses figures , tables etc.
  - Formal: created using precise syntax and meaning (formalism)
- Operational: describes intended system by describing its *desired behavior*.
- Descriptive: states the desired properties of the system in purely declarative fashion.

Specification

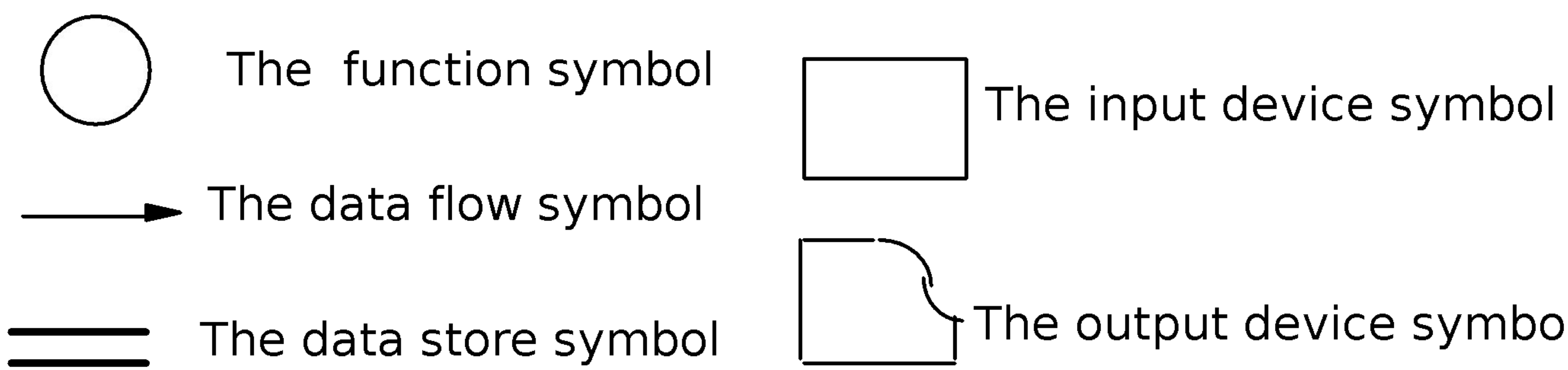
- Types of specifications
  - operational
    - Data Flow Diagrams
    - (Some) UML diagrams
    - Finite State Machines
    - Petri Nets
  - descriptive
    - Entity Relationship Diagrams
    - Logic-based notations
    - Algebraic notations

Data Flow Diagrams (DFDs)

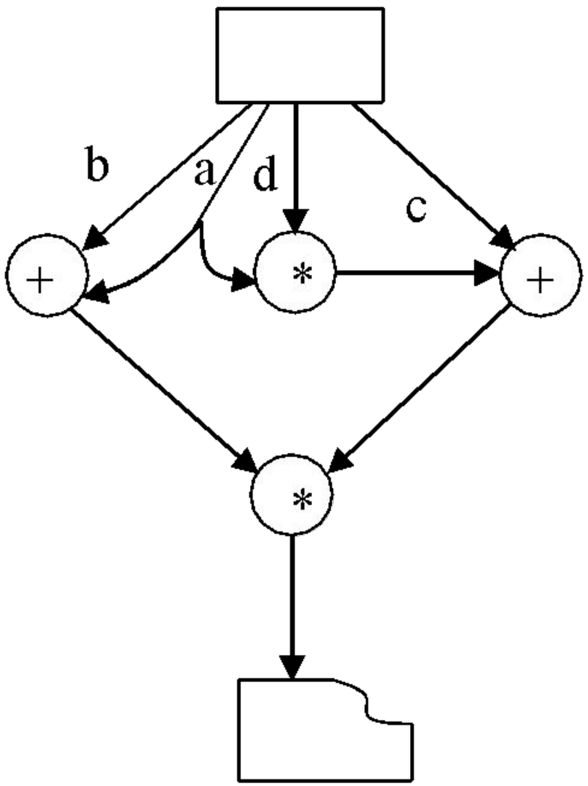
- A semi-formal operational specification
- System viewed as collection of processing steps or functions
- Represents how data flows through a sequence of processing steps
- DFDs have a graphical notation
- For example: Filtering of duplicate records in a customer database

Graphical notation

– *bubbles* represent functions
– *arcs* represent data flows
– *open boxes* represent persistent store
– *closed boxes* represent I/O



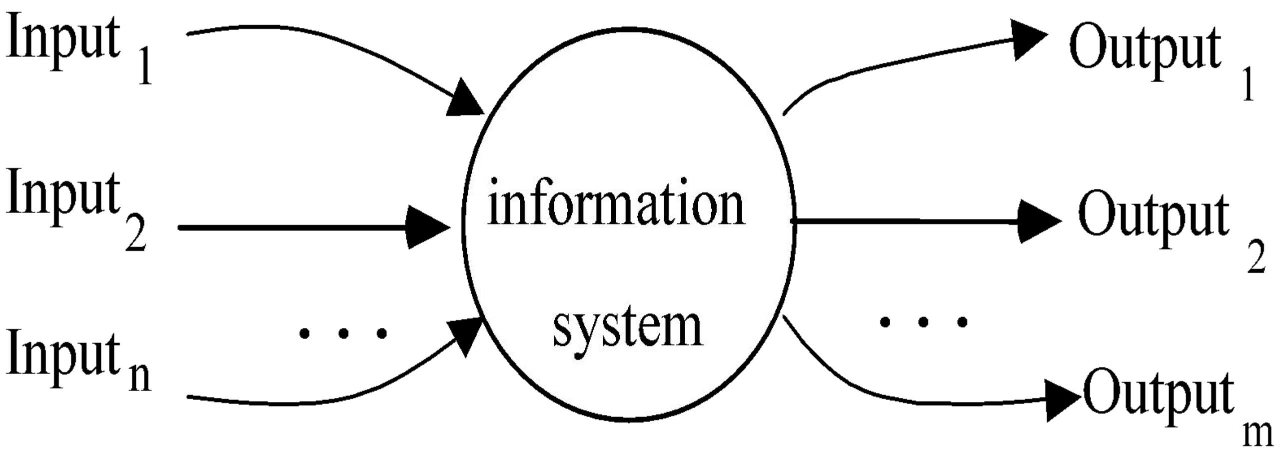
Example



specifies evaluation of
(a + b) * (c + a * d)

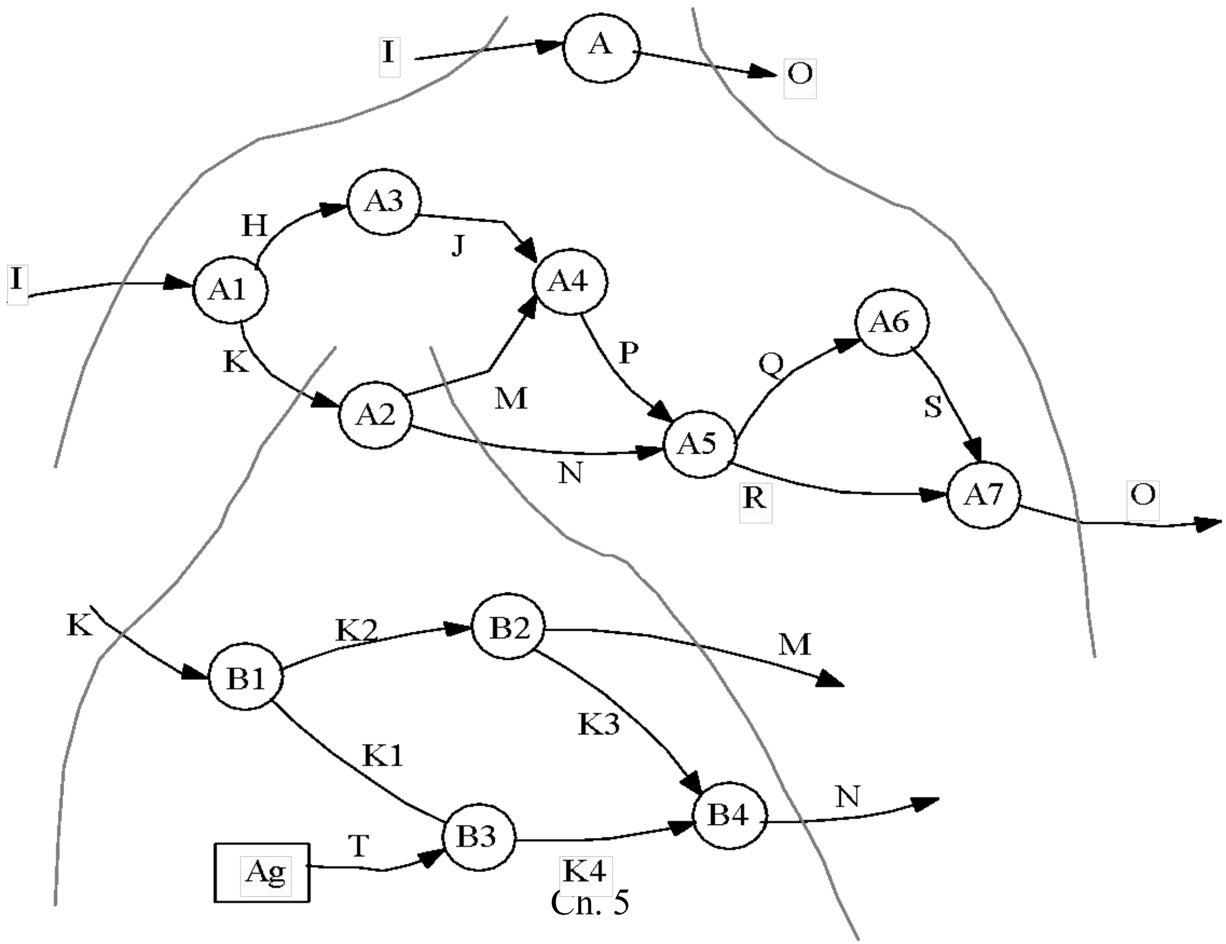
A construction "method" (1)

1. Start from the "context" diagram

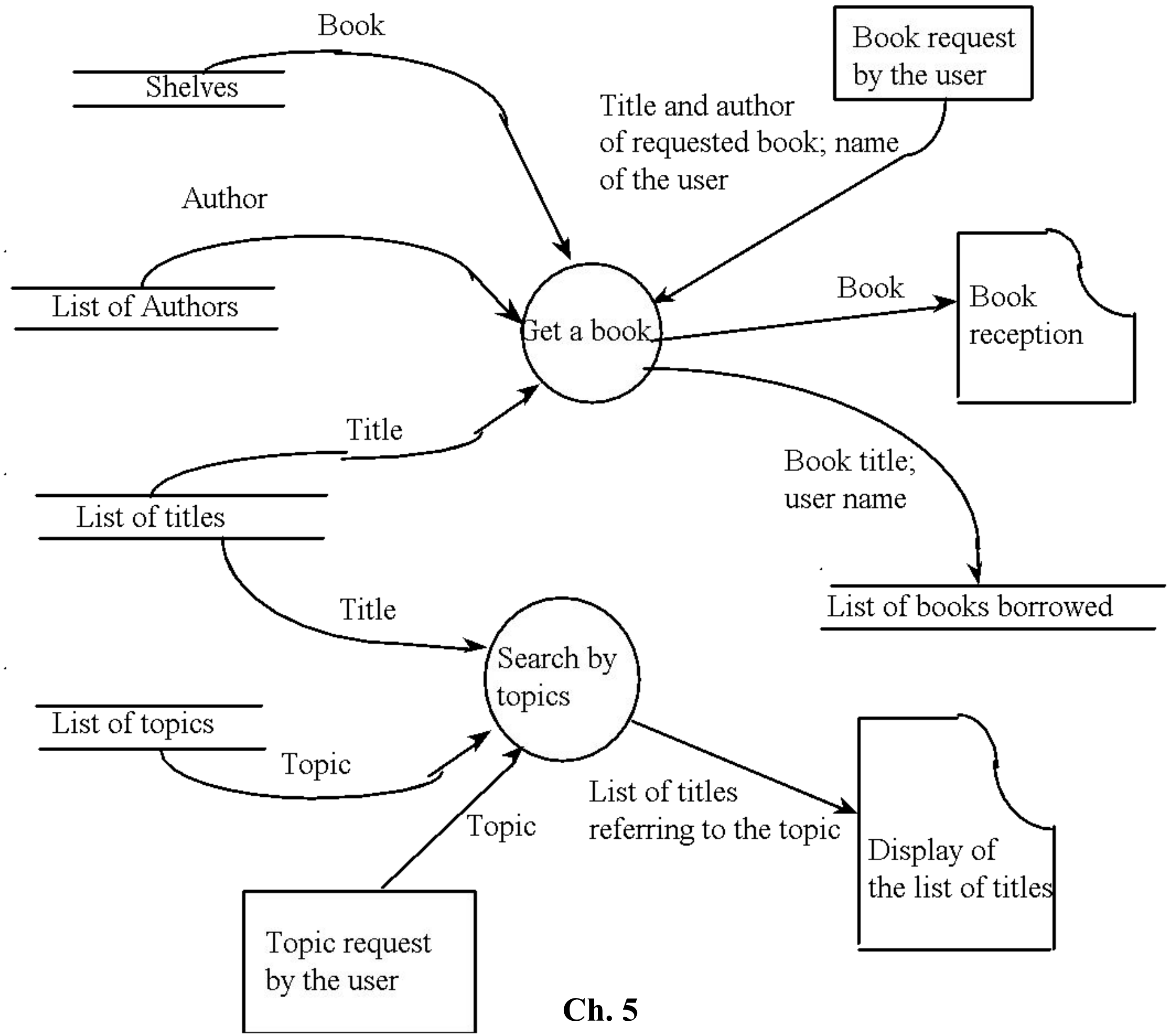


A construction "method" (2)

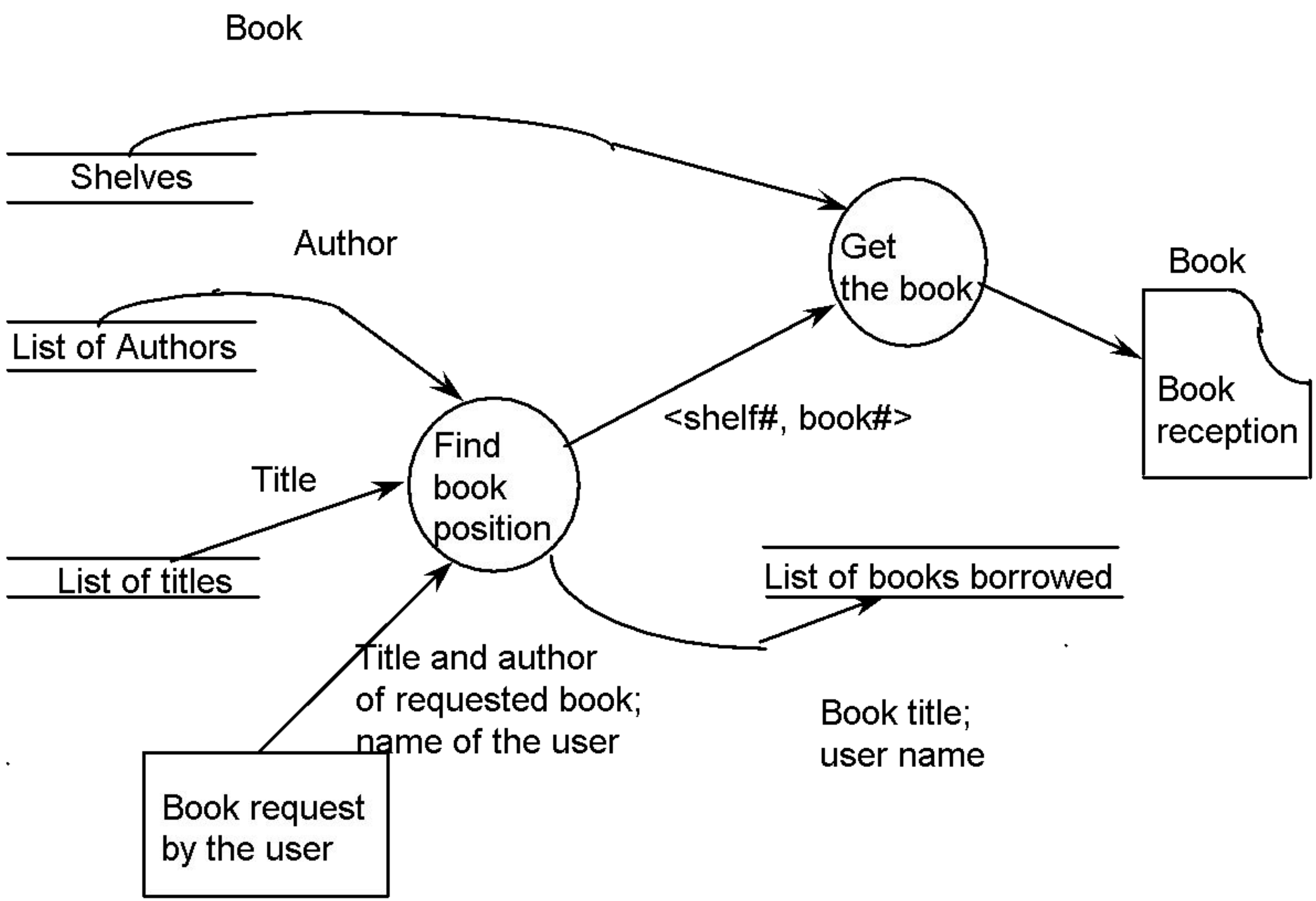
2. Proceed by refinements until you reach "elementary" functions



A library example (Level 0)

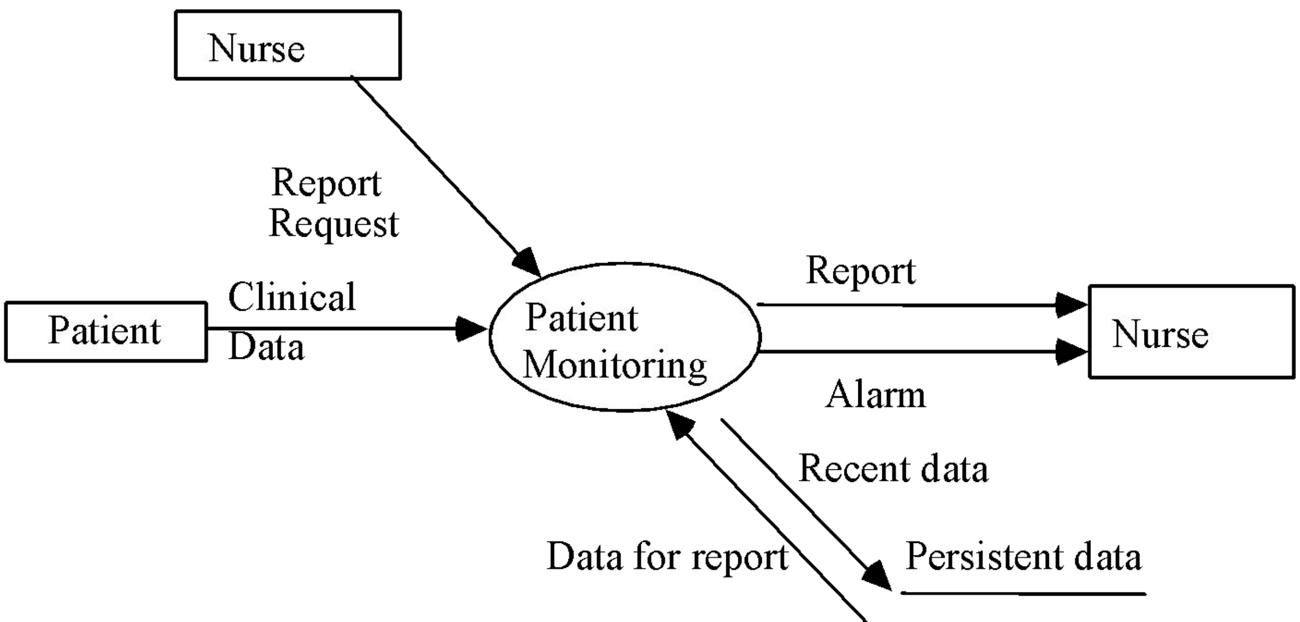


Refinement of "Get a book" (Level 1)

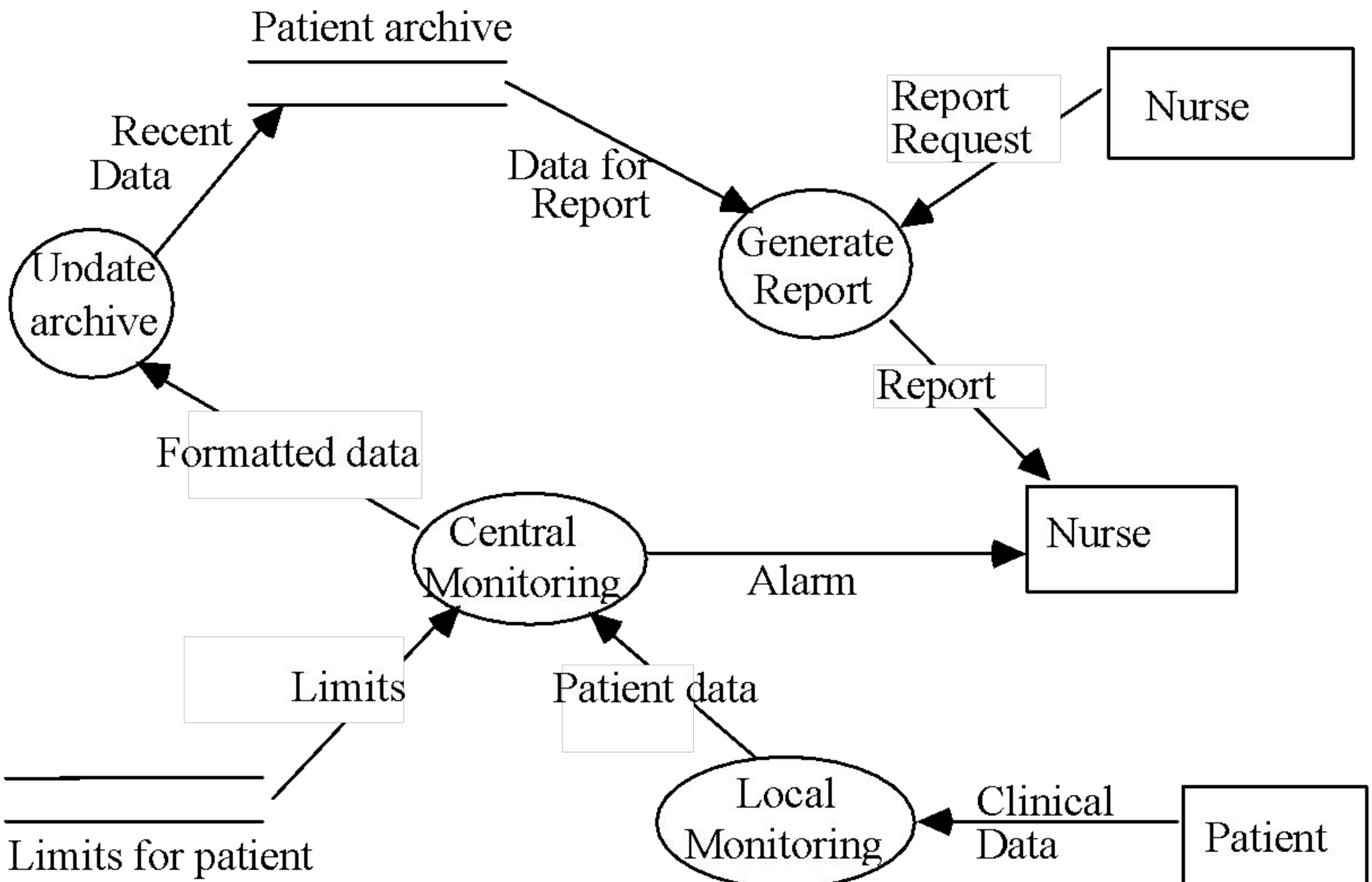


Patient monitoring systems

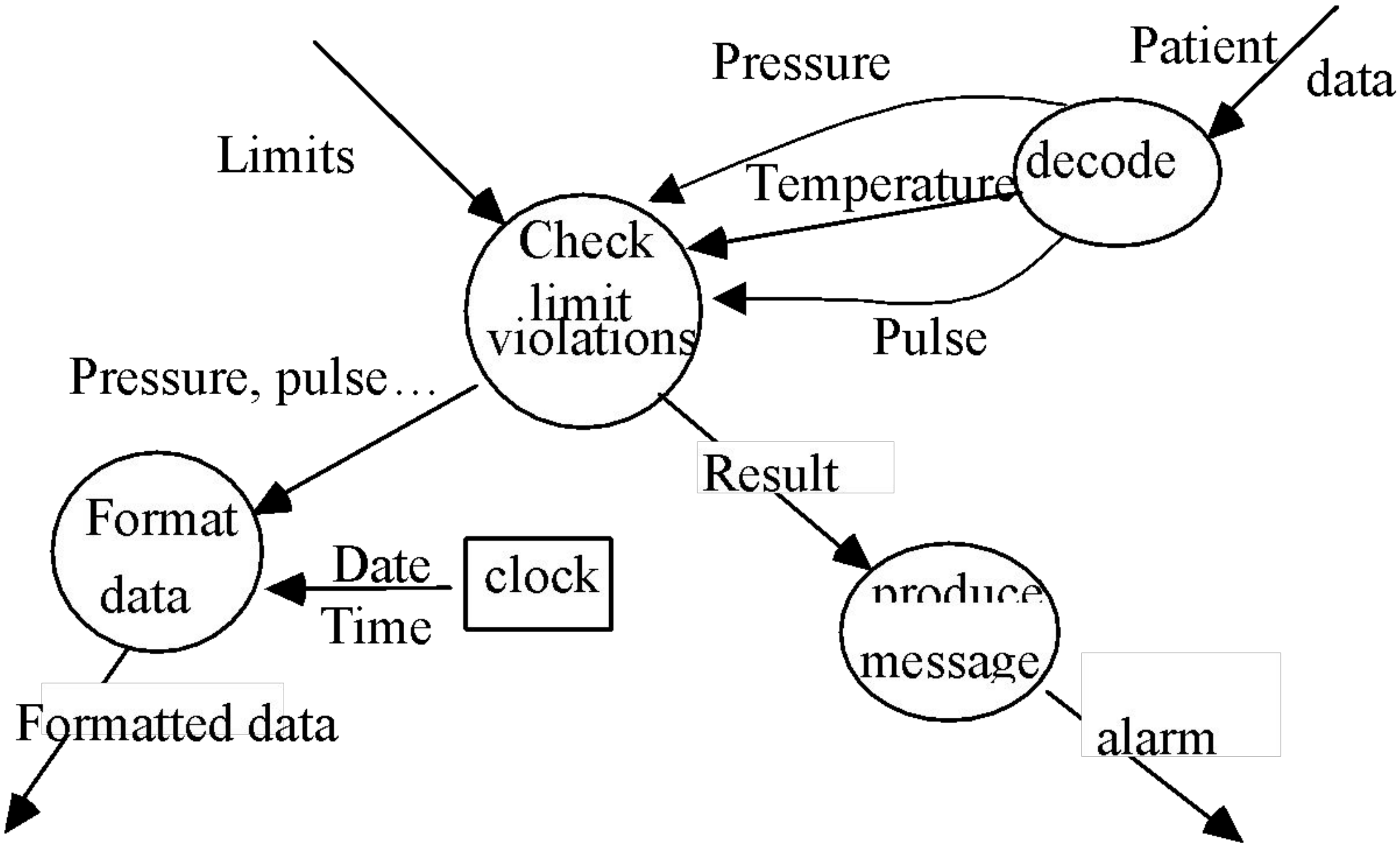
The purpose is to monitor the patients' vital factors--blood, pressure, temperature, …--reading them at specified frequencies from analog devices and storing readings in a DB. If readings fall outside the range specified for patient or device fails an alarm must be sent to a nurse. The system also provides reports.



A refinement



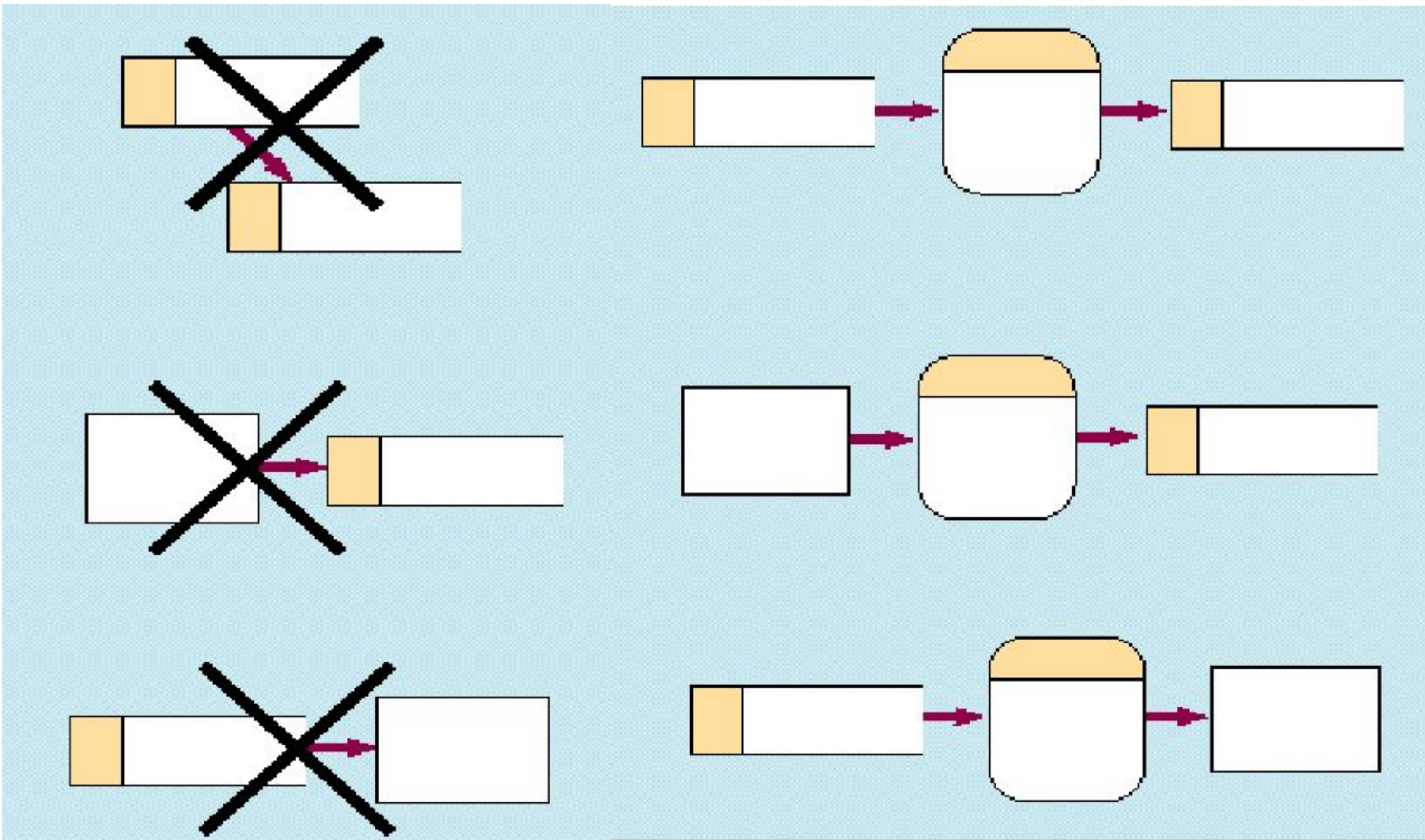
More refinement



Basic rules

- Balancing principle: Decomposed DFD (next lower level) should retain the same number of inputs and outputs from its previous higher level DFD
- No process can have only input(s)/output(s)
  - How to define leaf functions?
  - Inherent ambiguities

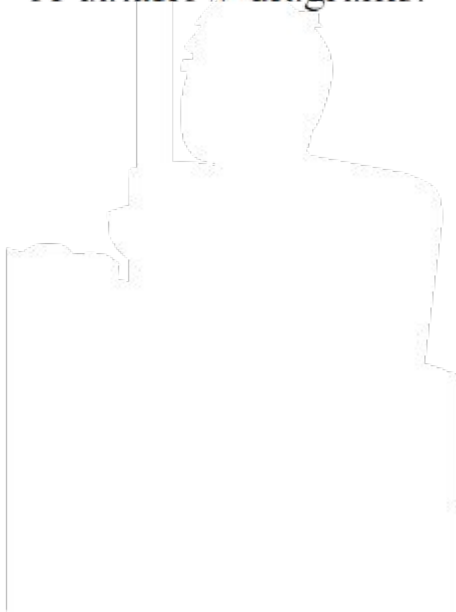
Basic rules



Creating DFDs (Lemonade Stand Example)

Example

The operations of a simple lemonade stand will be used **to demonstrate the creat**ion of dataflow diagrams.



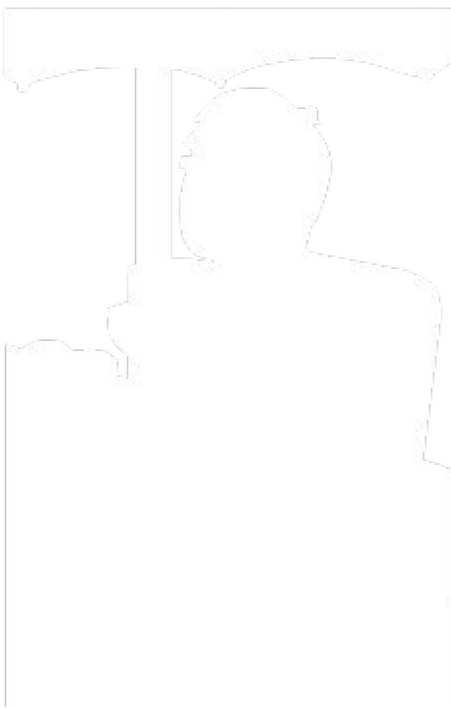
Steps:

1. Create a list of activities
2. Construct Context Level DFD
   (identifies sources and sink)
3. Construct Level 0 DFD
   (identifies manageable sub processes)
4. Construct Level 1- n DFD
   (identifies actual data flows and data stores)

Creating DFDs

Example

Think through the activities that take place at a lemonade stand.



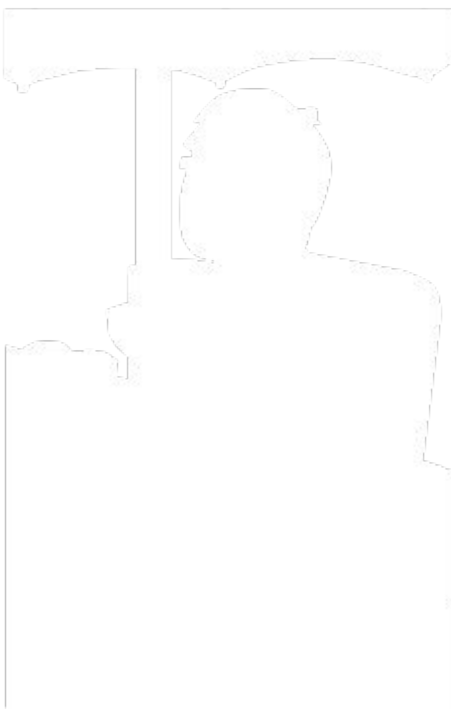
1. Create a list of activities

Customer Order
Serve Product
Collect Payment
Produce Product
Store Product

Creating DFDs

Example

Also think of the additional activities needed to support the basic activities.



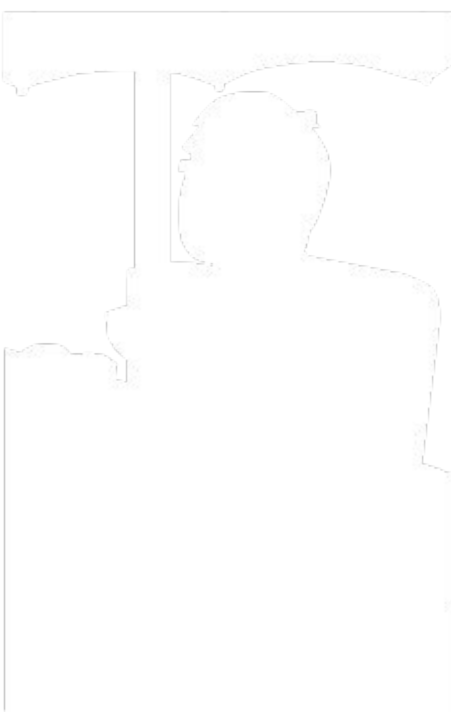
1. Create a list of activities

Customer Order
Serve Product
Collect Payment
Produce Product
Store Product
Order Raw Materials
Pay for Raw Materials
Pay for Labor

Creating DFDs

Example

Group these activities in some logical fashion, possibly functional areas.



1. Create a list of activities

Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

Creating DFDs

Example

Create a context level diagram identifying the sources and sinks (users).

Customer Order
Serve Product
Collect Payment

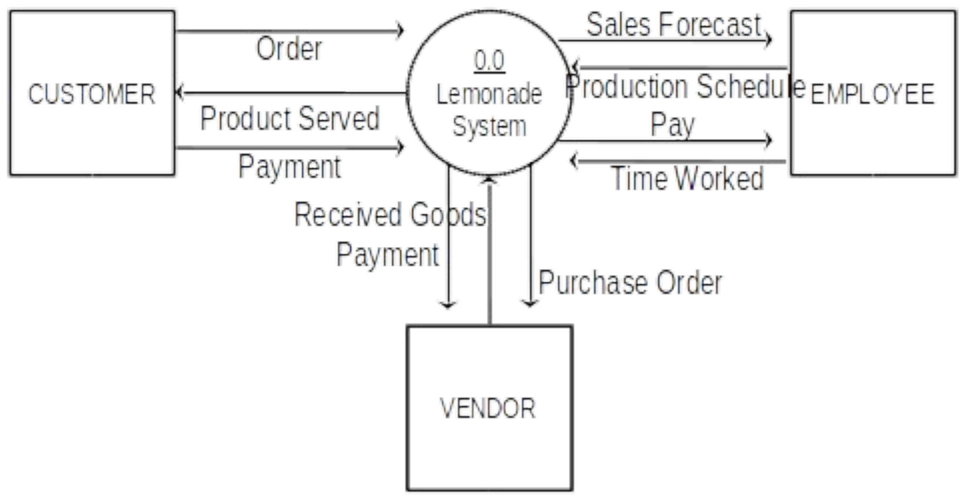
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

2. Construct Context Level DFD (identifies sources and sink)

Context Level DFD



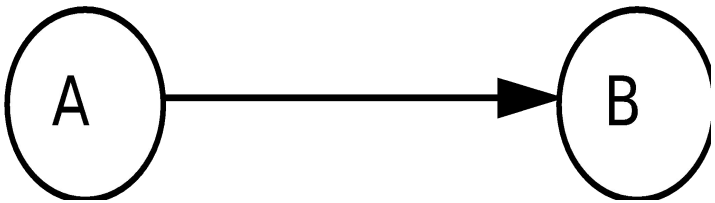
Identify manageable subprocesses and refine the DFD

Drawbacks

- Time consuming
- It does not provide a complete picture of the system and sometimes leaves vital physical entities.
- A DFD can be confusing and programmers might not differentiate between its levels.

Drawbacks

- Control information is absent



Possible interpretations:

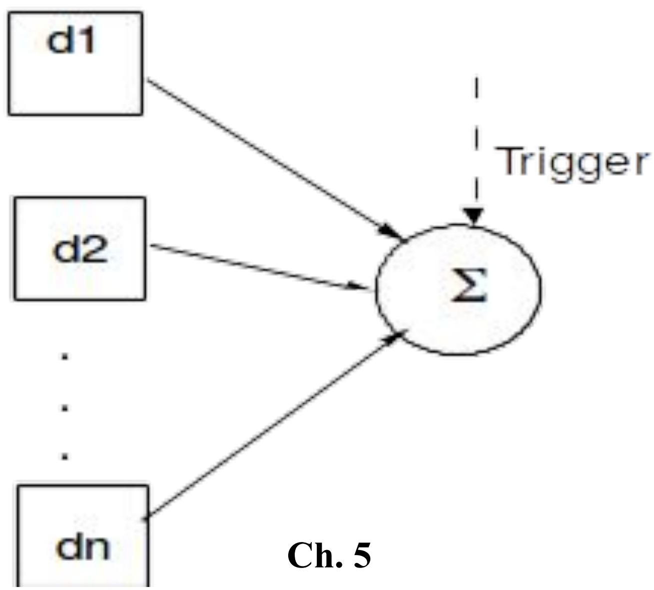
(a) A produces datum, waits until B consumes it

(b) B can read the datum many times without consuming it

(c) a pipe is inserted between A and B

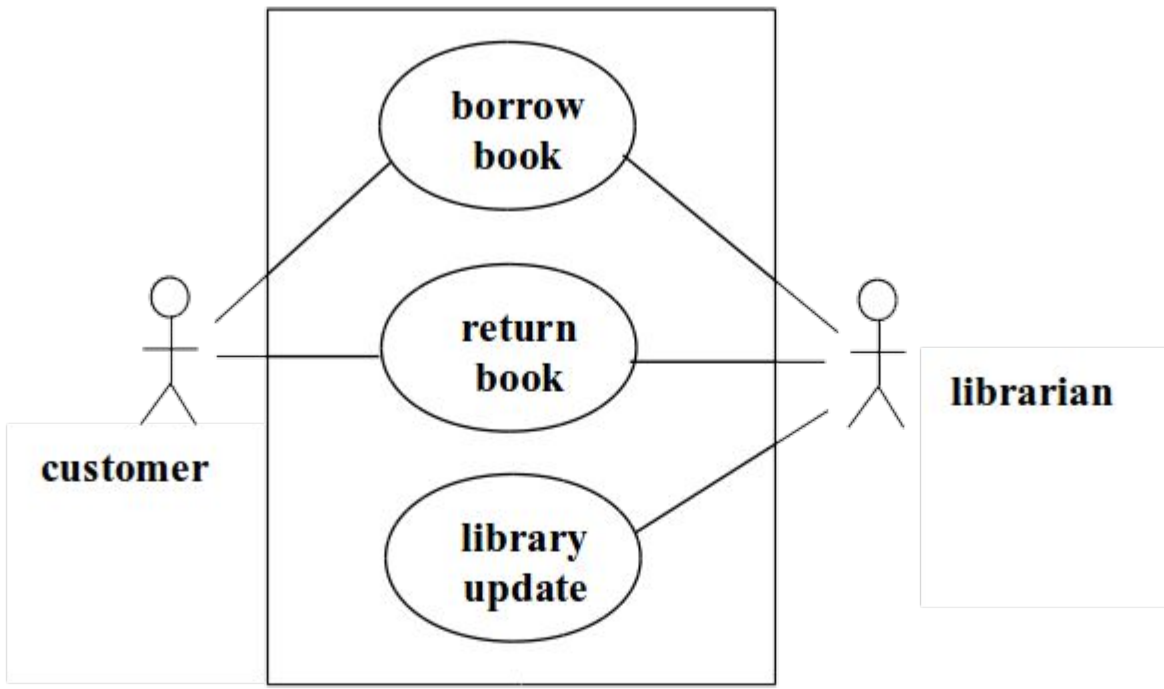
Solutions

- Formalizations: There have been attempts to *formalize* DFDs
- There have been attempts to *extend* DFDs (e.g., for real-time systems)



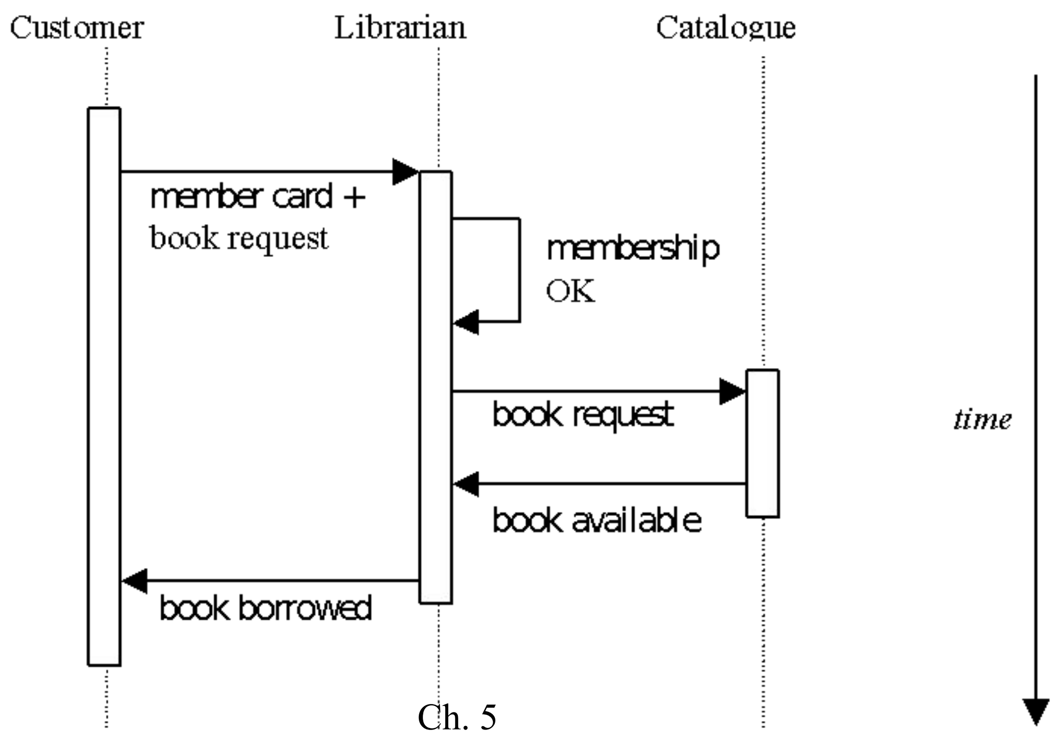
UML use-case diagrams

- Define functions on basis of actors and actions



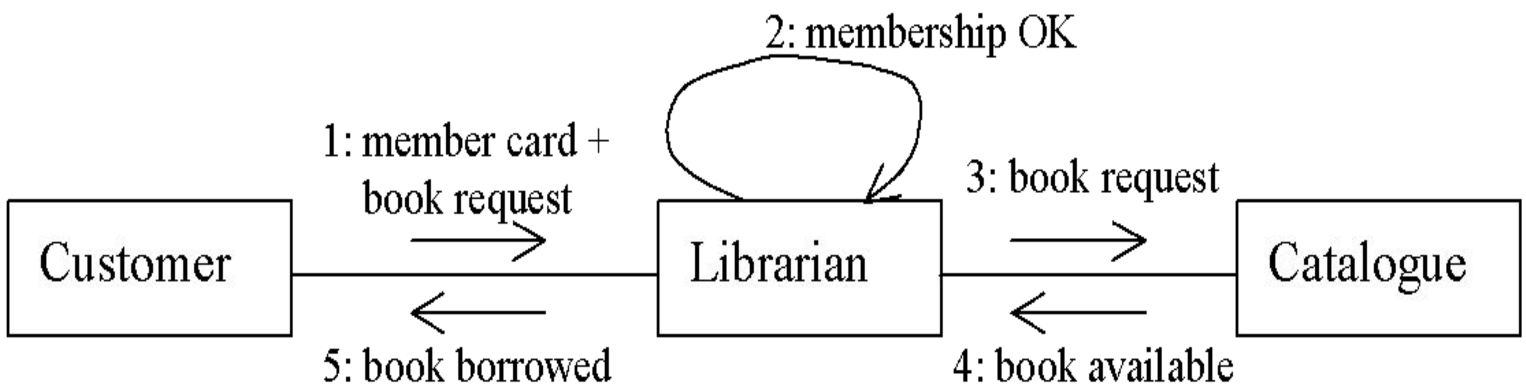
UML sequence diagrams

- Describe how objects interact by exchanging messages
- Provide a dynamic view



UML collaboration diagrams

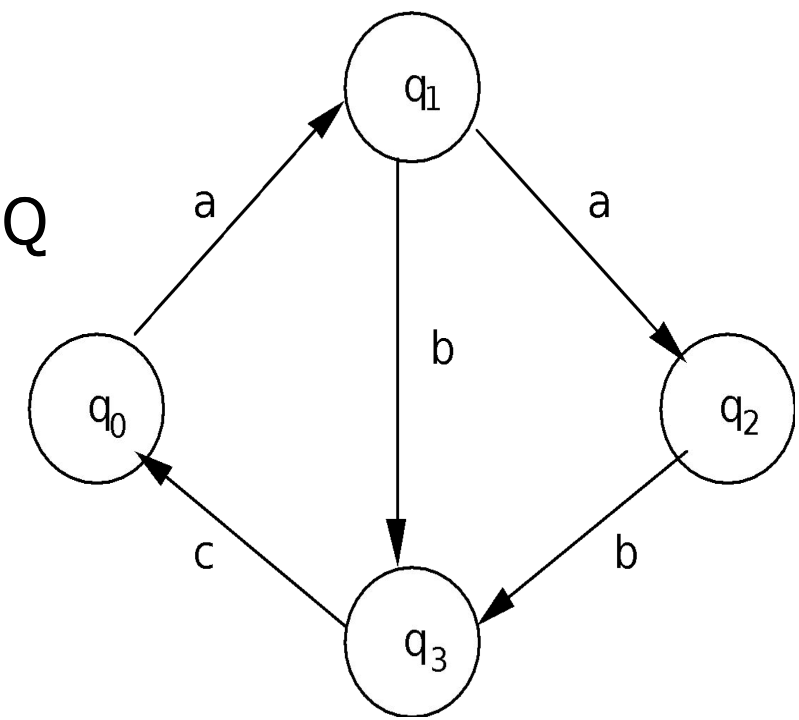
- Give object interactions and their order
- Equivalent to sequence diagrams



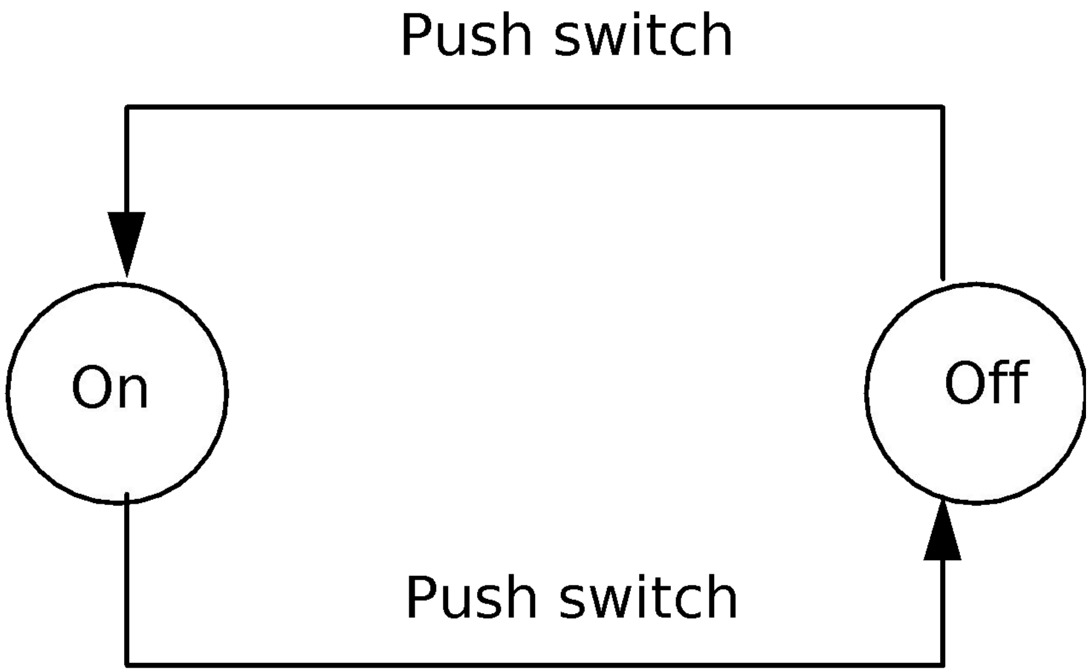
Finite state machines (FSMs)

- Can specify control flow aspects
- Defined as

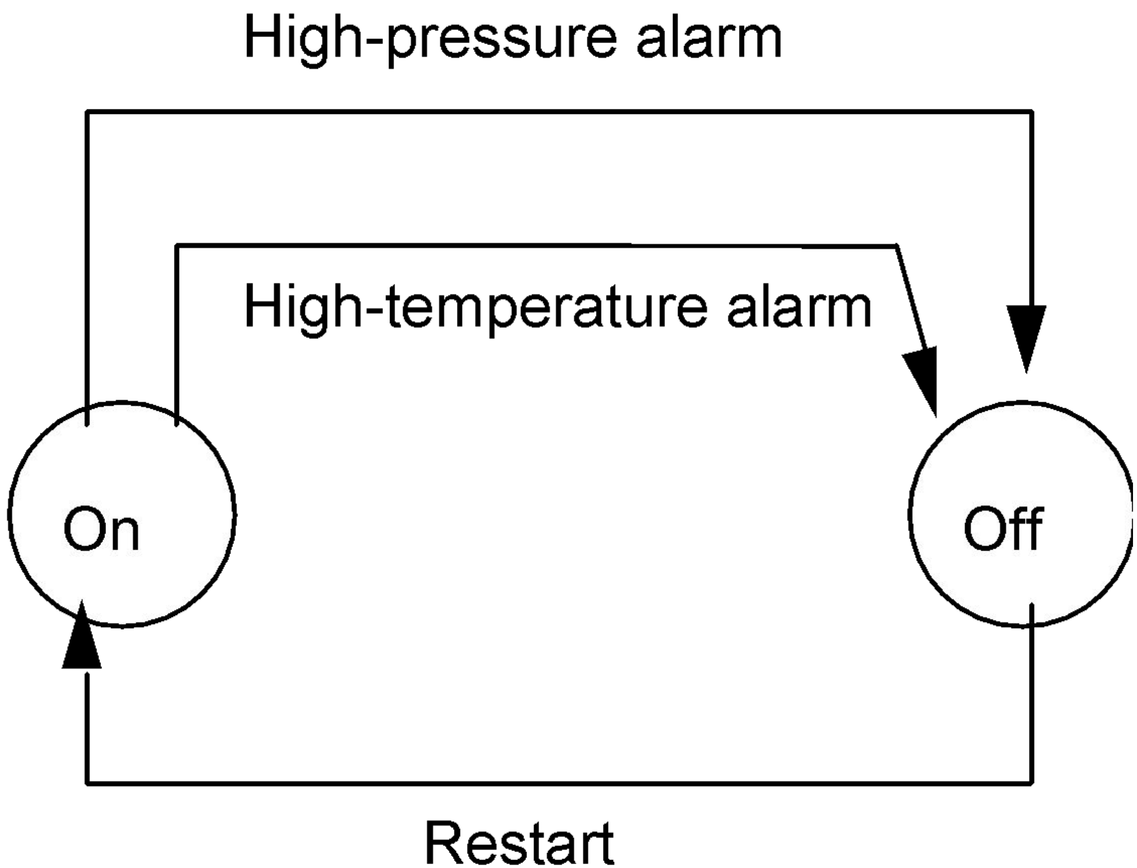
a finite set of states, Q;
a finite set of inputs, I;
a transition function d : Q x I Q
(d can be a partial function)



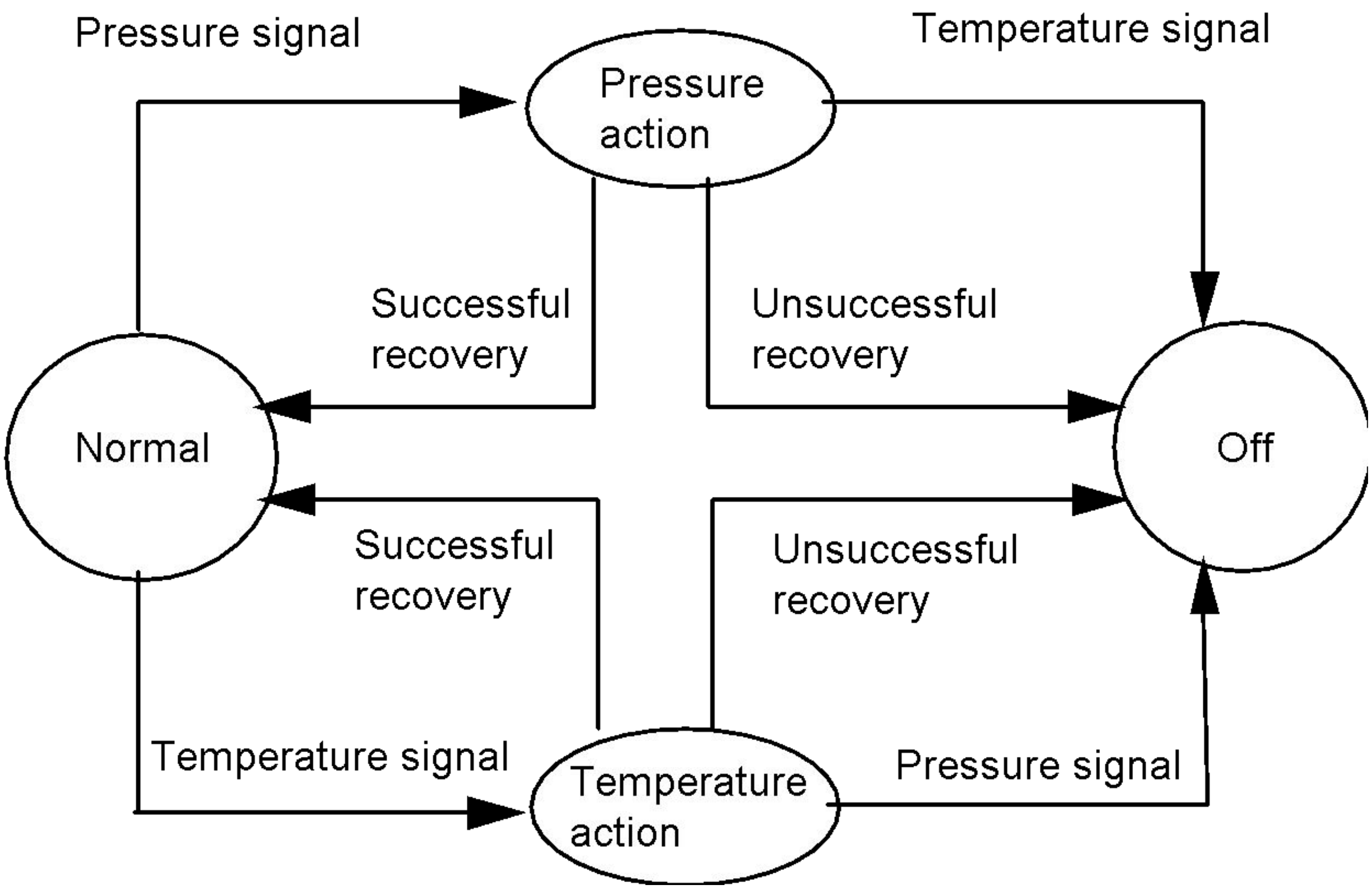
Example: a lamp



Another example: a plant control system



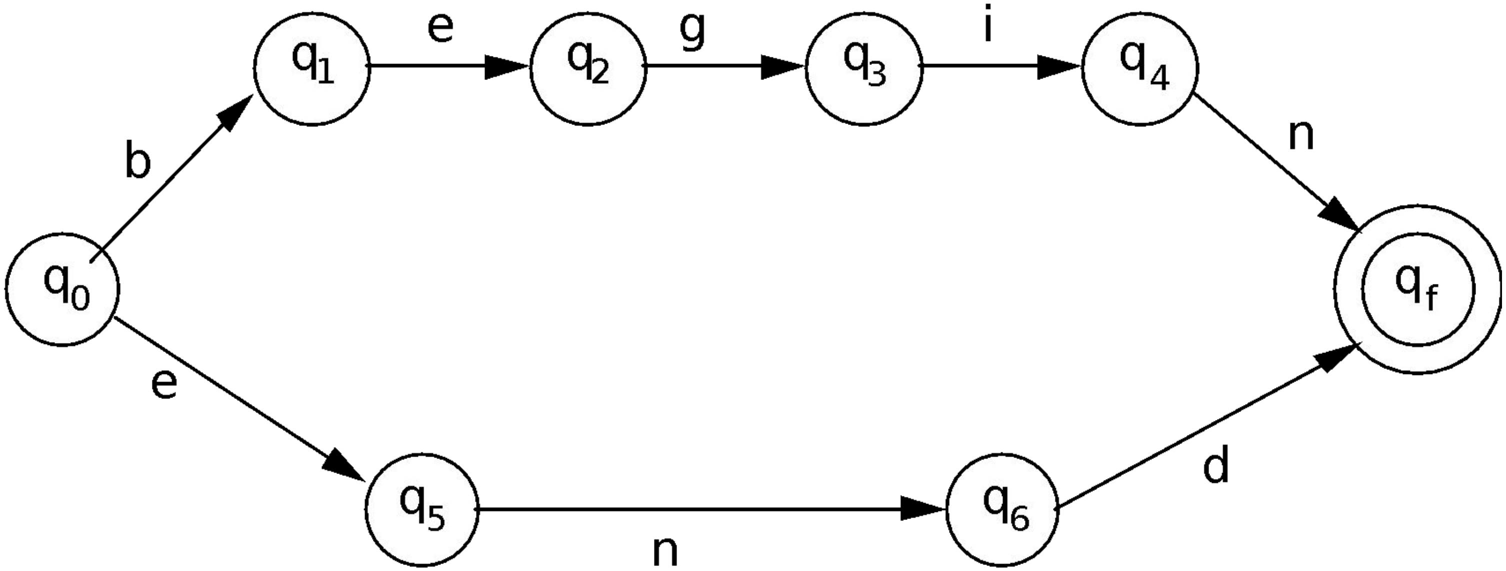
A refinement



Classes of FSMs

- Deterministic/nondeterministic
- FSMs as recognizers
  - introduce final states
- FSMs as transducers
  - introduce set of outputs

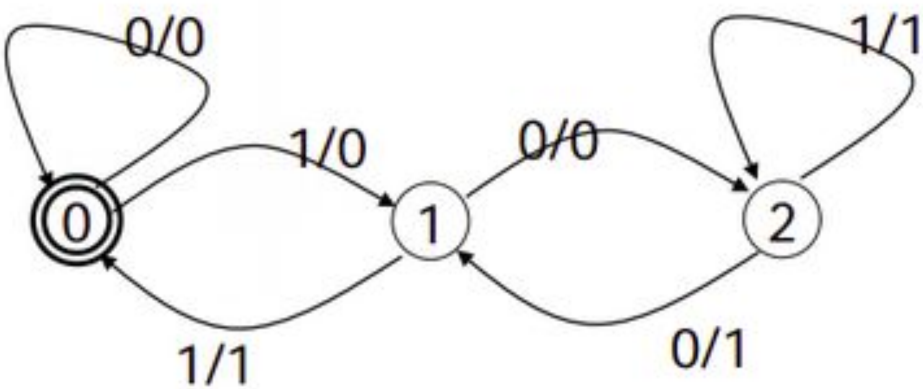
FSMs as recognizers



q_f is a final state

FSMs as transducers

input	output
0	0
11	01
110	010
1001	0011
1100	0100
1111	0101
10010	00110



Petri net

- Also known as place/transition (PT) net, is one of the several mathematical modeling languages for the description of distributed systems.

- It is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, represented by bars) and places (i.e. conditions, represented by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows).

Petri net

A quadruple (P,T,F,W)

P: places T: transitions (P, T are finite)

F: flow relation ($F \subseteq \{P \times T\} \cup \{T \times P\}$)

W: weight function ($W: F \rightarrow N - \{0\}$)

Properties:

(1) $P \cap T = \emptyset$

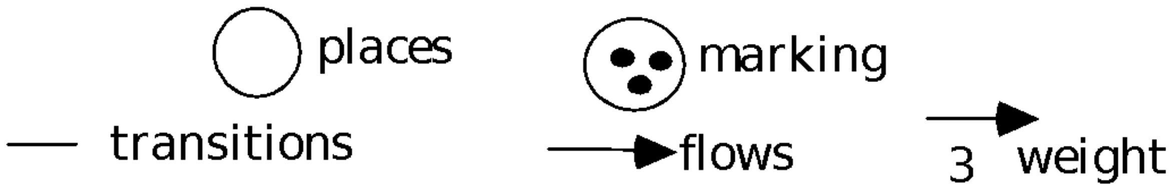
(2) $P \cup T \neq \emptyset$

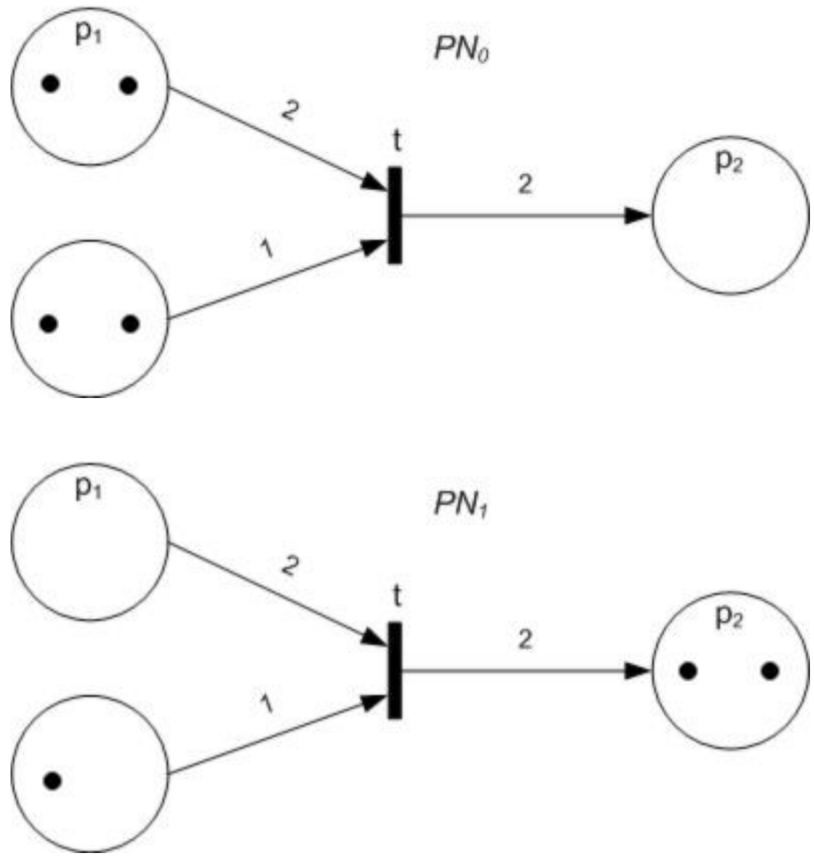
(3) $F \subseteq (P \times T) \cup (T \times P)$

(4) $W: F \rightarrow N$ is a multiset of arcs, i.e. it assigns to each arc a non-negative integer arc multiplicity (or weight) Default value of W is 1

State defined by marking: $M: P \rightarrow N$

Graphical representation





Semantics

- Transition t is enabled iff
  - $\forall p \in$ t's input places, $M(p) \geq W(<p,t>)$
- t fires: produces a new marking M' in places that are either t's input or output places or both
  - if p is an input place: $M'(p) = M(p) - W(<p,t>)$
  - if p is an output place: $M'(p) = M(p) + W(<t,p>)$
  - if p is both an input and an output place: $M'(p) = M(p) - W(<p,t>) + W(<t,p>)$

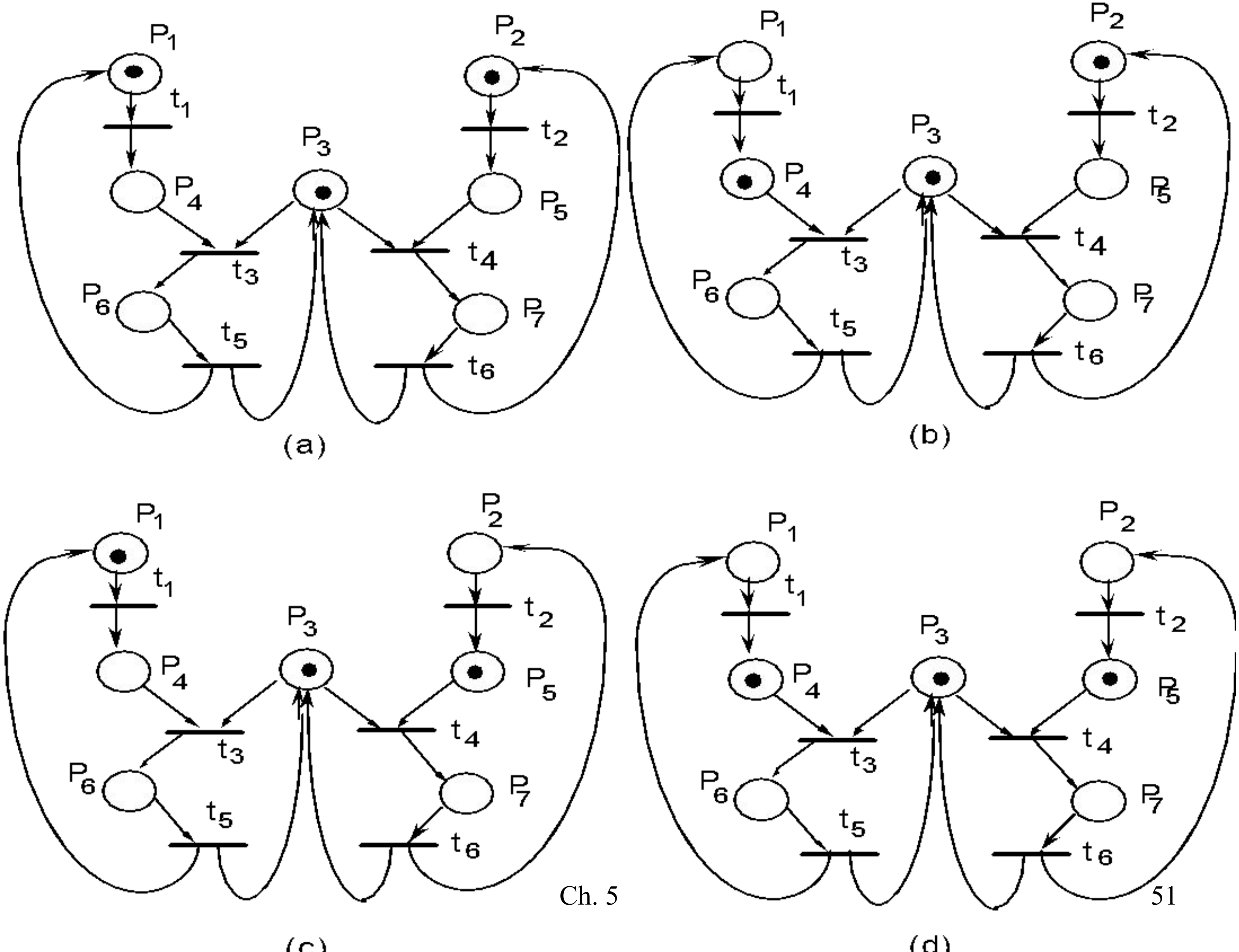
Modeling with Petri nets

- Places represent distributed states
- Transitions represent actions or events that may occur when system is in a certain state
- They can occur as certain conditions hold on the states

Nondeterminism

- Any of the enabled transitions may fire
- Model does not specify which fires, nor when it fires

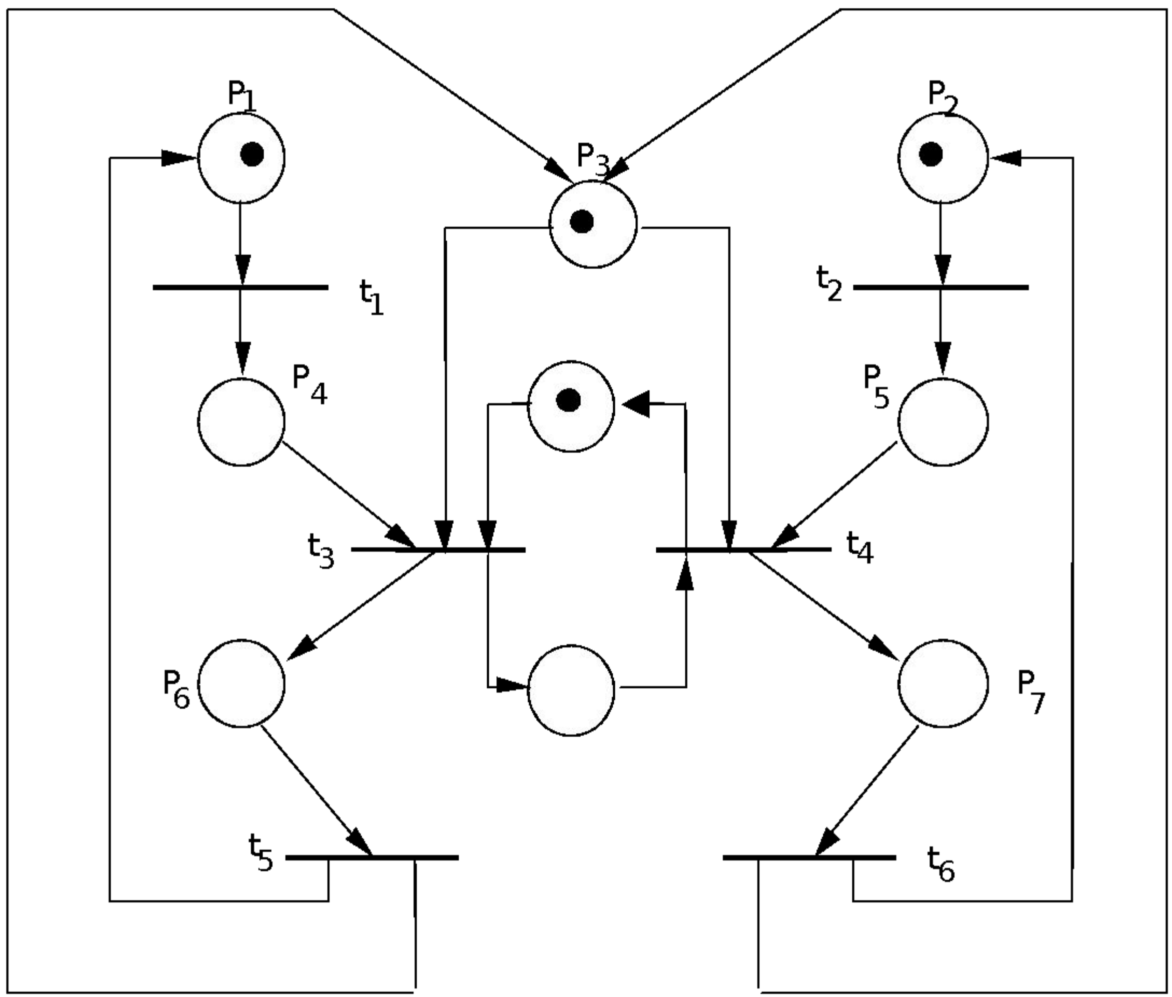
after (a) either (b) or (c) may occur, and then (d)



Common cases

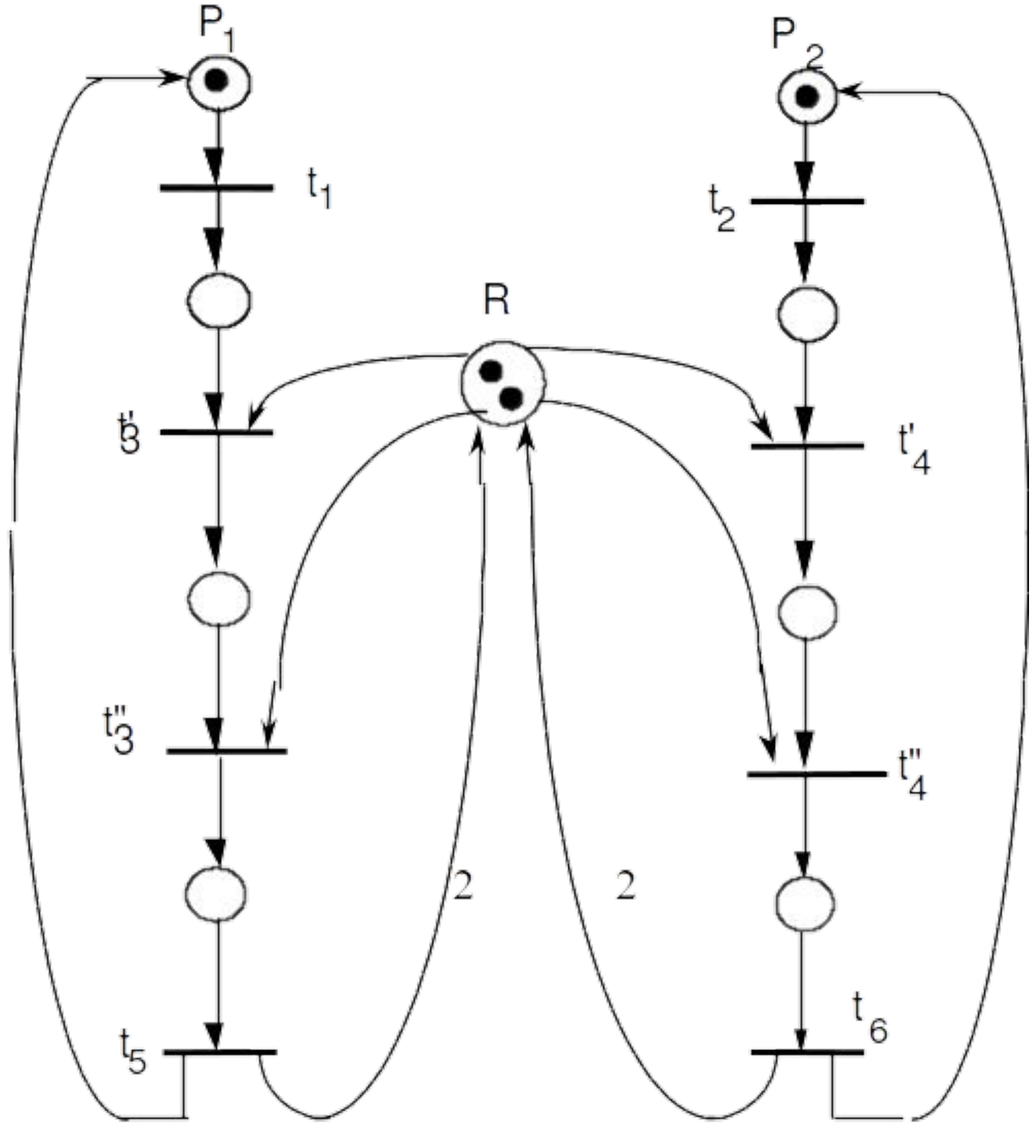
- Concurrency
  - two transitions are enabled to fire in a given state, and the firing of one does not prevent the other from firing
    - see t_1 and t_2 in case (a)
- Conflict
  - two transitions are enabled to fire in a given state, but the firing of one prevents the other from firing
    - see t_3 and t_4 in case (d)
    - place P_3 models a shared resource between two processes
  - no policy exists to resolve conflicts (known as *unfair* scheduling)
  - a process may never get a resource (*starvation*)

How to avoid starvation



imposes alternation

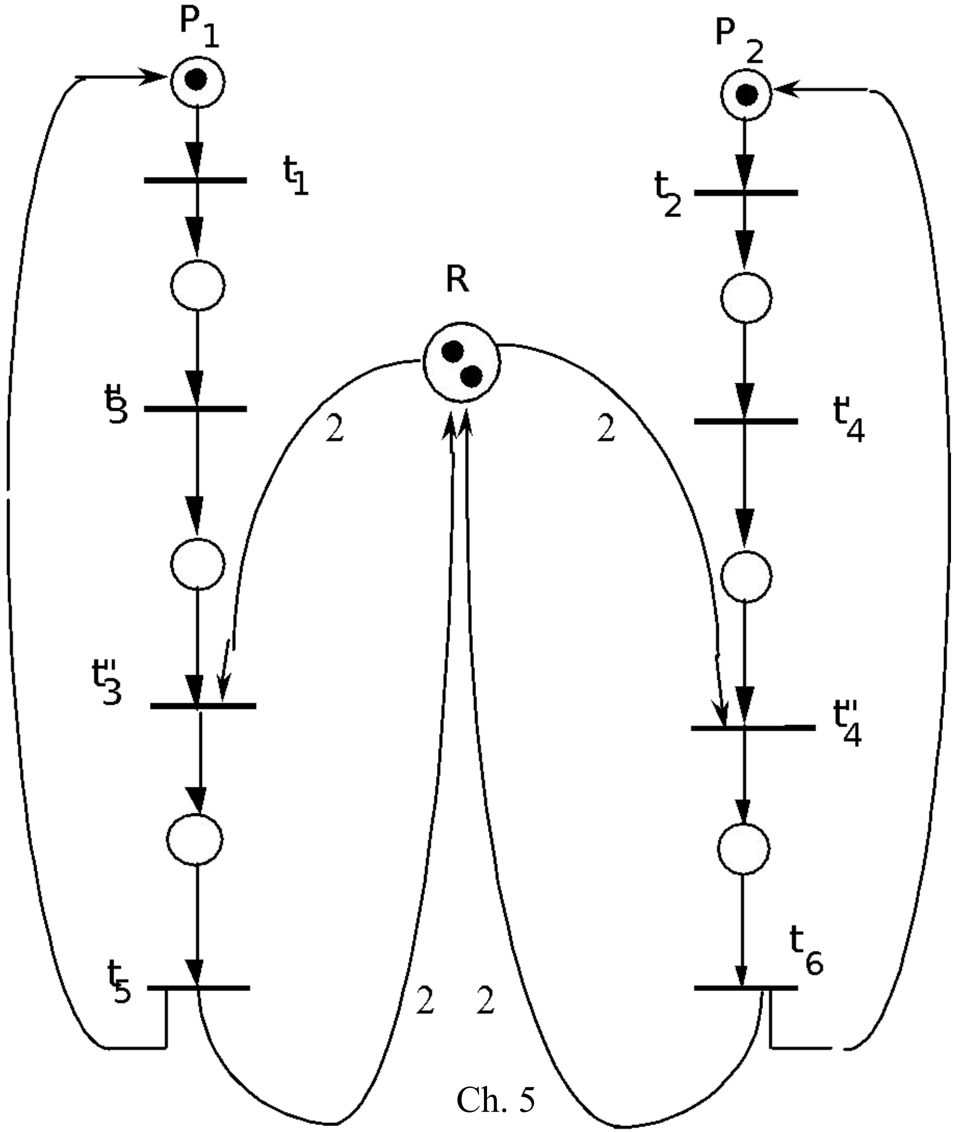
Deadlock issue



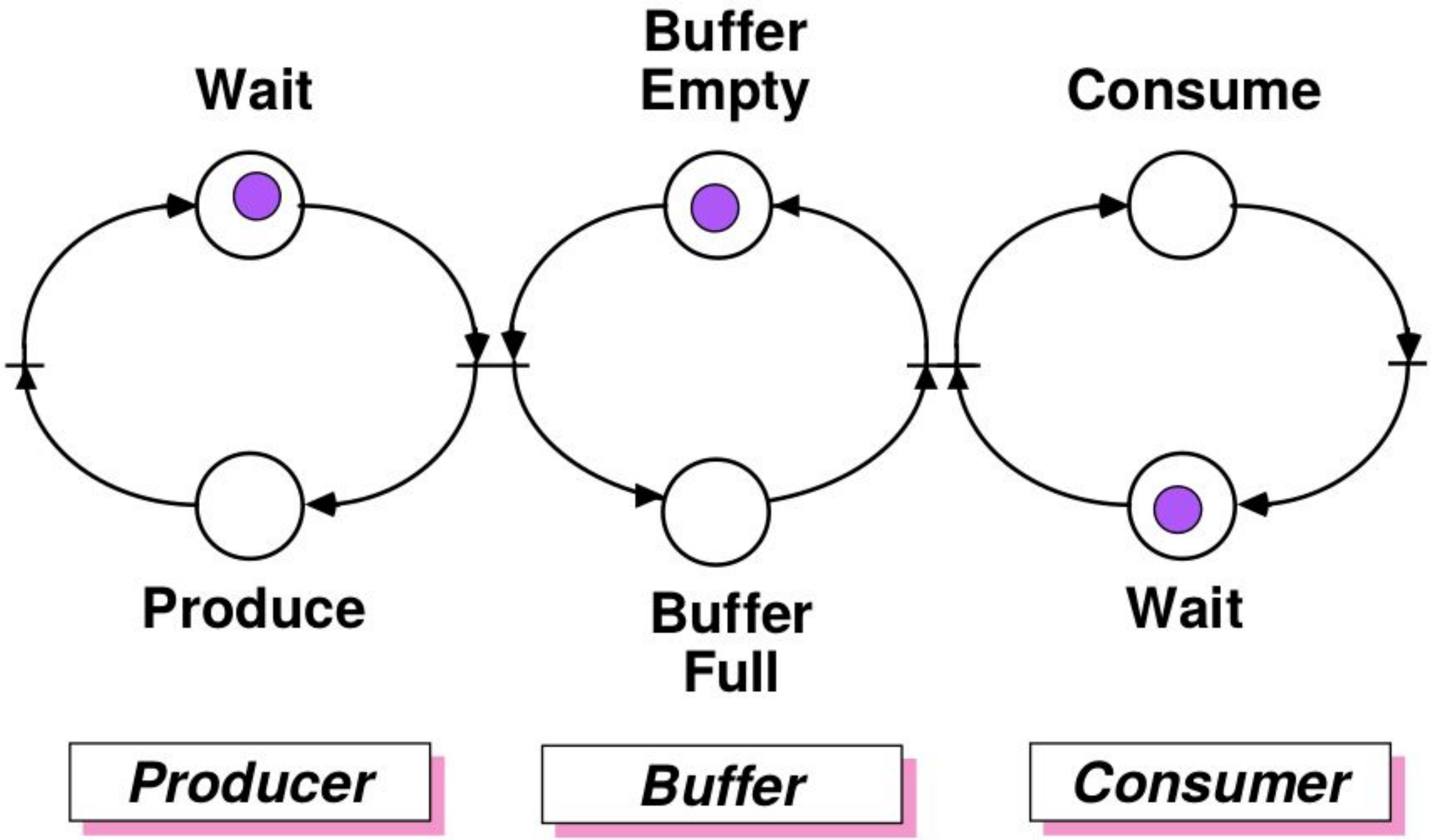
consider

$< t_1, t'_3, t_2, t'_4 >$

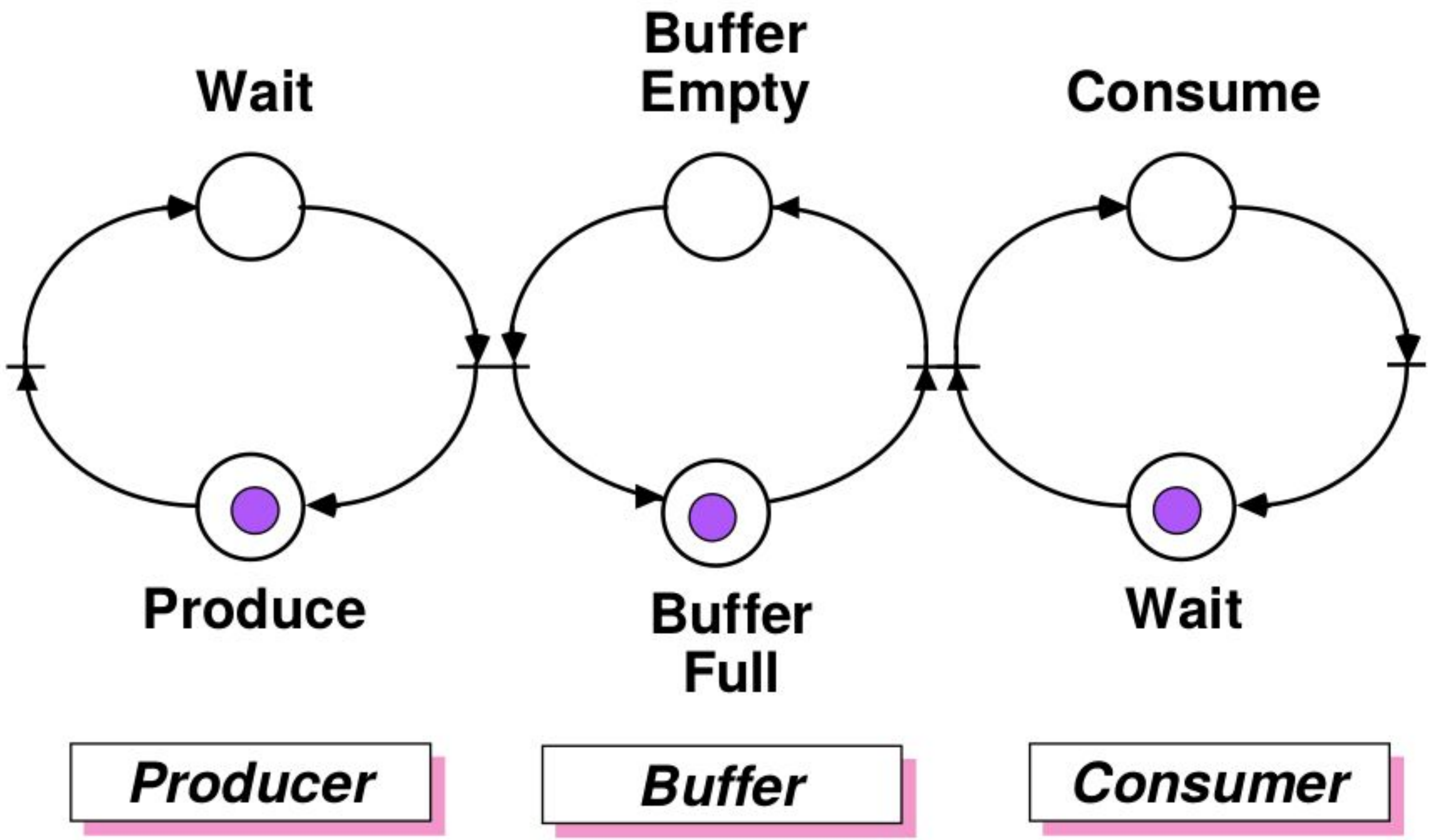
A deadlock-free net



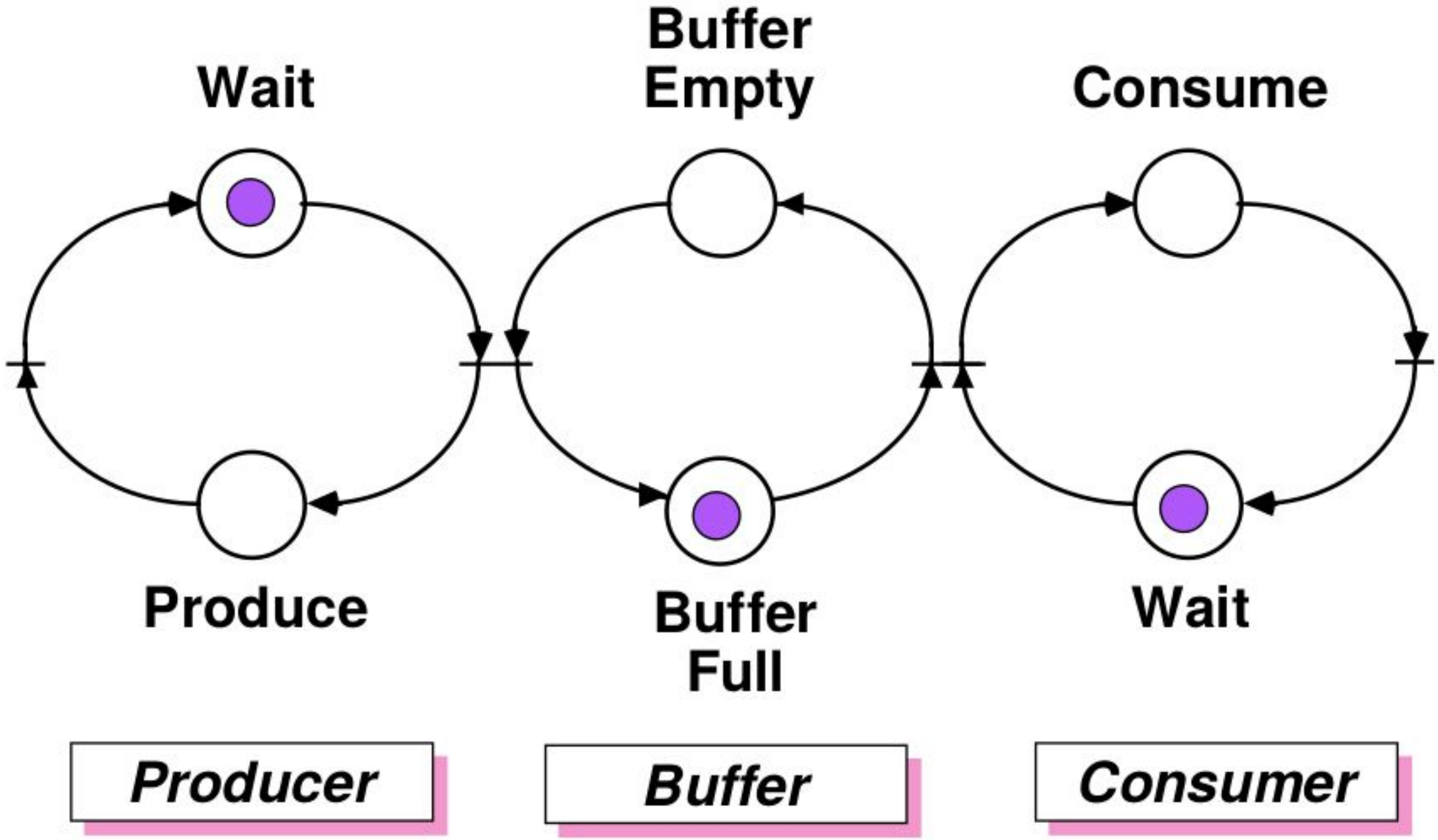
Producer-consumer example (1)



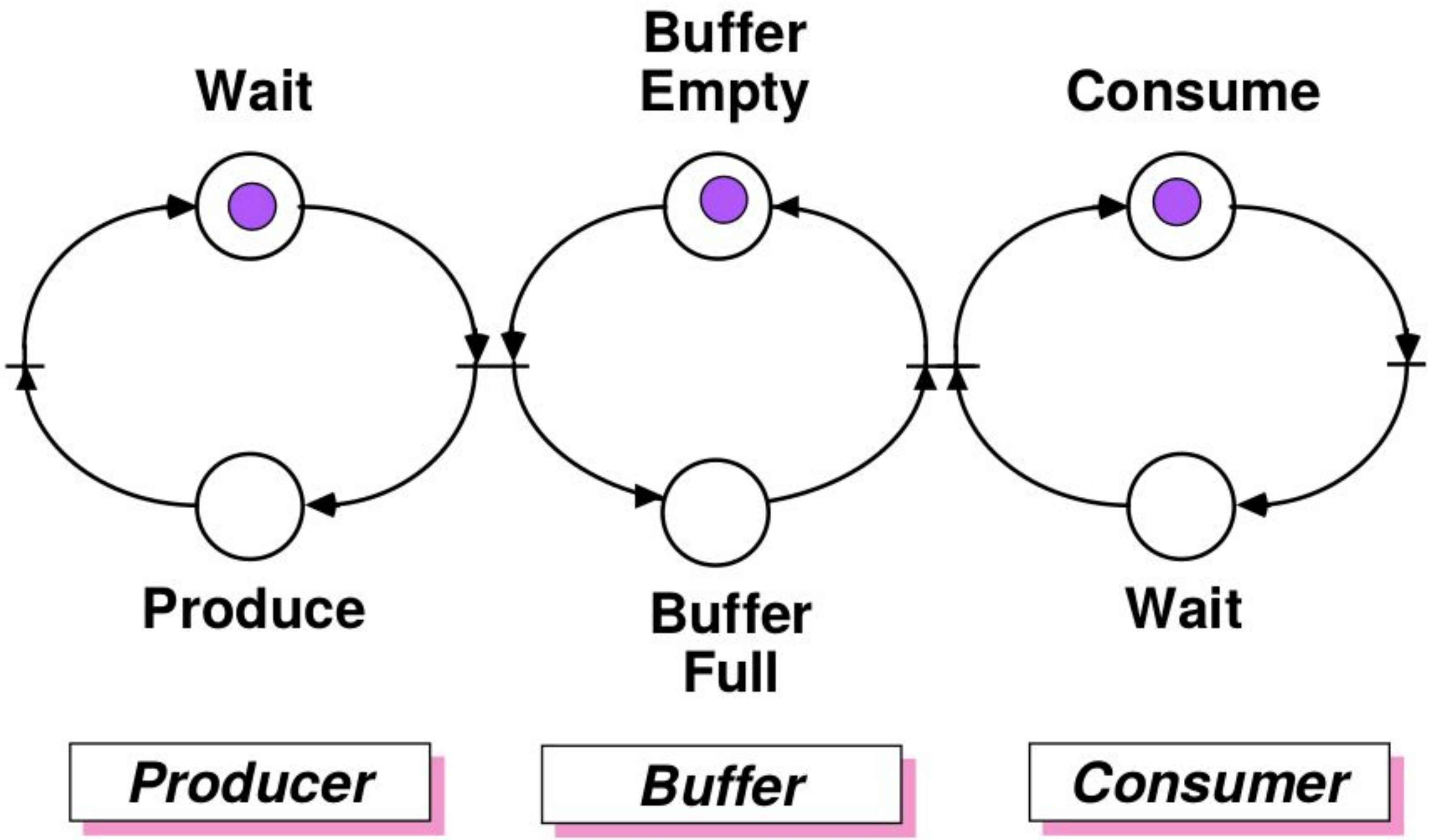
Producer-consumer example (2)



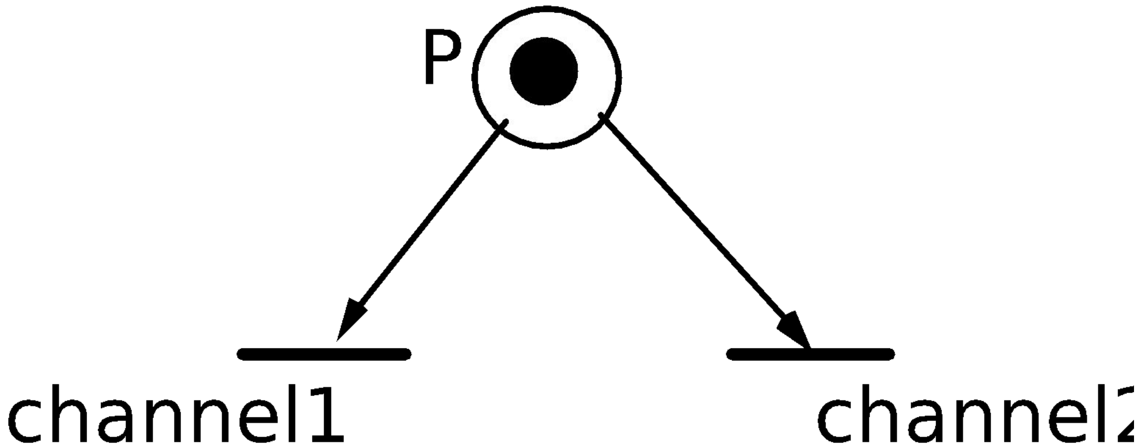
Producer-consumer example (3)



Producer-consumer example (4)



Limitations and extensions

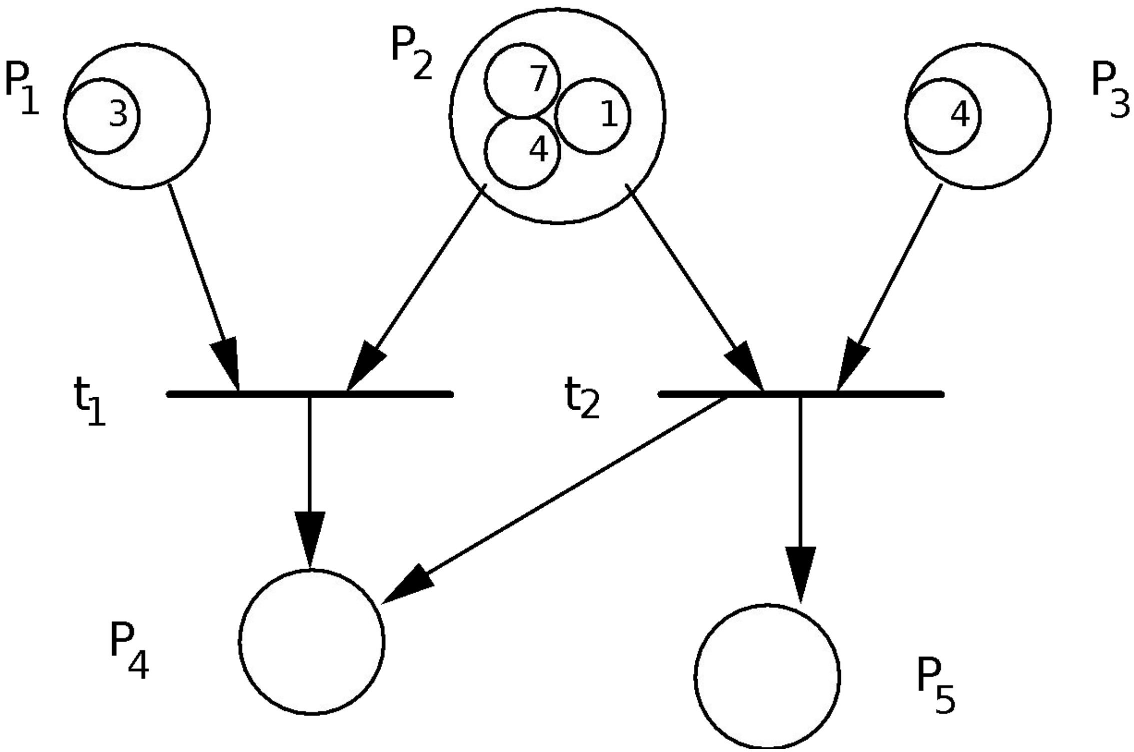


Token represents a message.
You wish to say that the delivery channel depends on contents.
How?
Petri nets cannot specify selection policies.

Extension 1
assigning values to tokens

- Transitions have associated predicates and functions
- Predicate refers to values of tokens in input places selected for firing
- Functions define values of tokens produced in output places

Example



Predicate P2 > P1 and function P4 := P2 + P1 associated with t1

Predicate P3 = P2 and functions P4 := P3 - P2 and P5 := P2 + P3 are associated with t2

The firing of t1 by using <3,7> would produce the value 10 in P4. t2 can then fire using <4, 4>

Extension 2
specifying priorities

- A priority function pri from transitions to natural numbers:
- pri: T N
- When several transitions are enabled, only the ones with maximum priority are allowed to fire
- Among them, the one to fire is chosen nondeterministically

Extension 3
Timed Petri nets

- A pair of constants <tmin, tmax> is associated with each transition
- Once a transition is enabled, it must wait for at least tmin to elapse before it can fire
- If enabled, it *must* fire before tmax has elapsed, unless it is disabled by the firing of another transition before tmax

Declarative specifications

ER diagrams

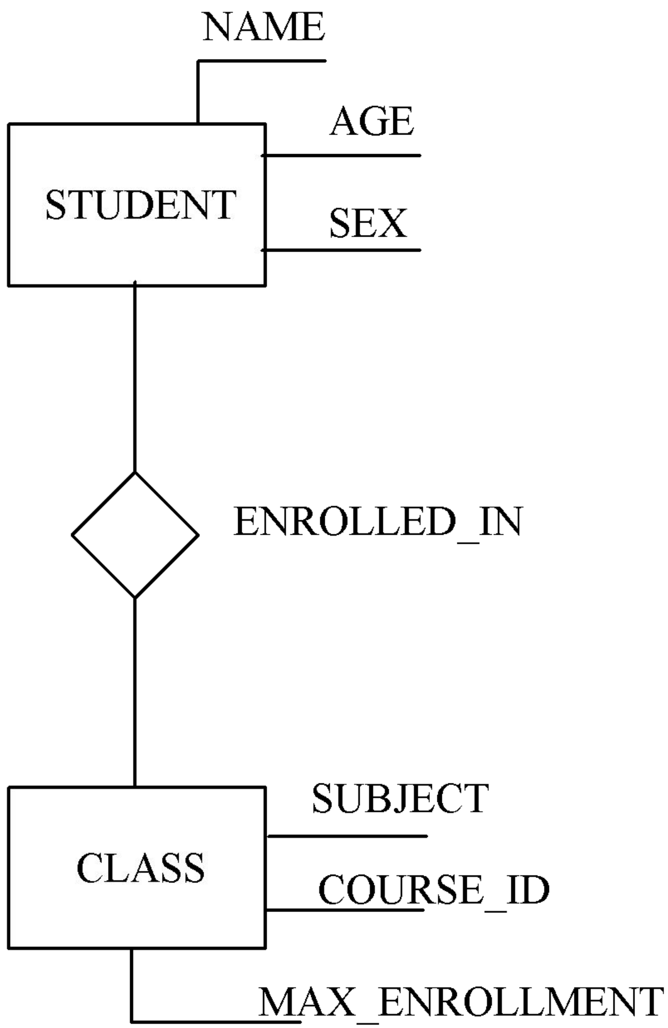
Logic specifications

Algebraic specifications

ER diagrams

- Often used as a complement to DFD to describe conceptual data models
- Based on entities, relationships, attributes
- They are the ancestors of class diagrams in UML

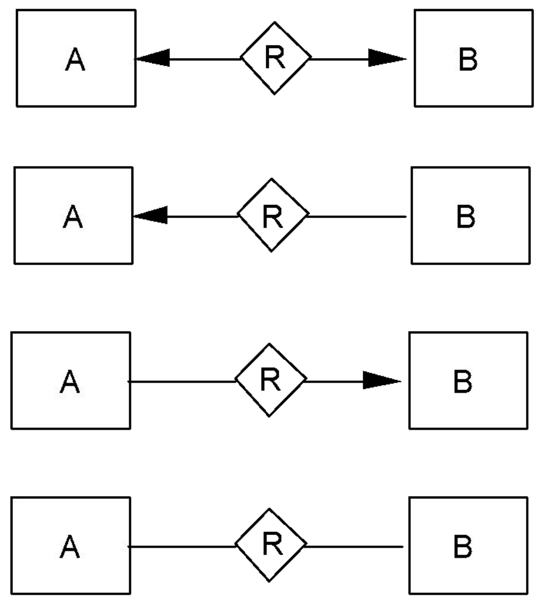
Example



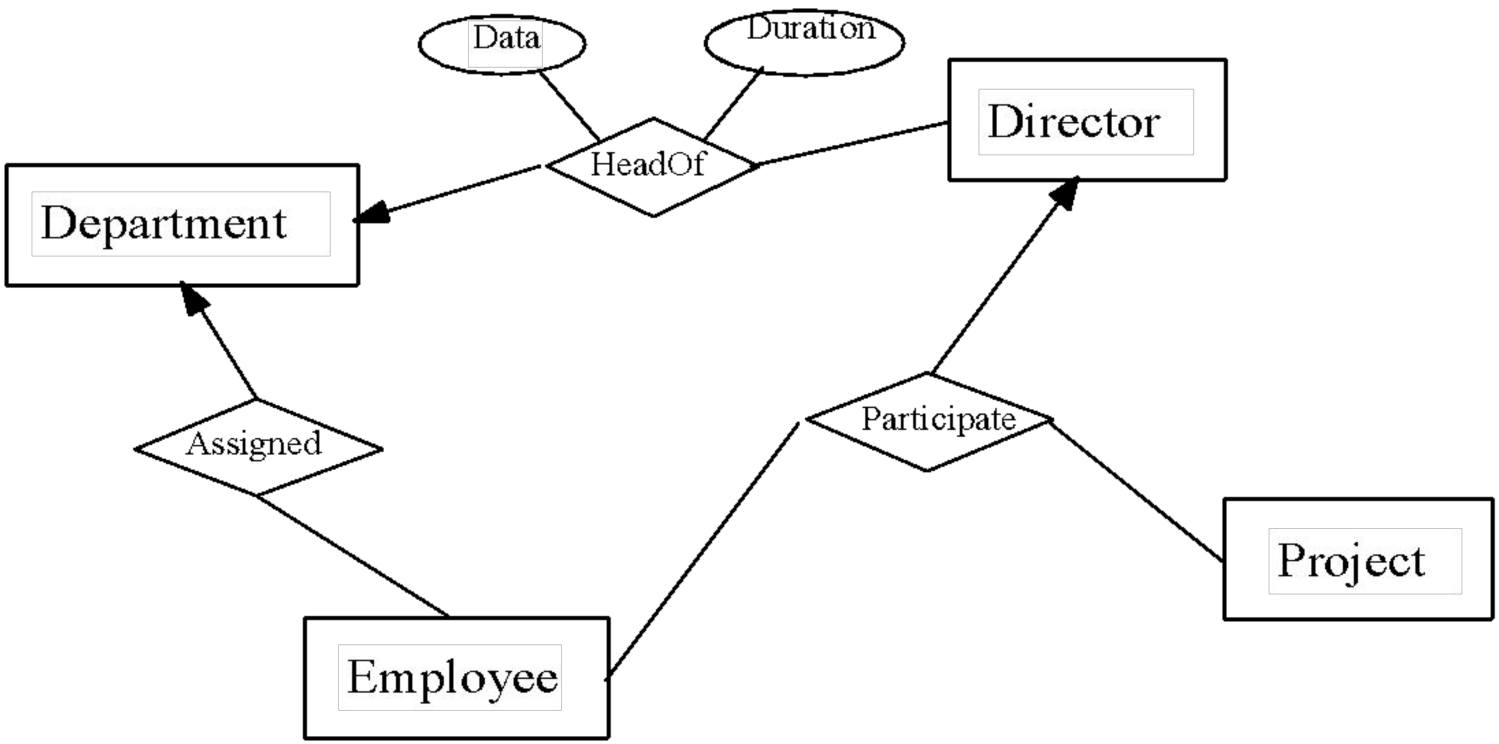
NAME
AGE
STUDENT
SEX
ENROLLED_IN
SUBJECT
CLASS
COURSE_ID
MAX_ENROLLMENT

Relations

- Relations can be partial
- They can be annotated to define
  - one to one
  - one to many
  - many to one
  - many to many



Non binary relations



Logic Specification techniques

Testing and Proofs

Testing	Proof
Observable Property	Any program property
Verify program for one execution	Verify program for all executions
Manual development with automated regression	Manual development with automated proof checkers
Most practical approach now	Practical for small critical programs

- So why learn about proofs if they are not practical?
  - Proofs tell us how to think about program correctness
  - Foundation for static analysis tools

How would you argue that this program is correct?

```
float sum(float *array, int length) {
float sum = 0.0;
int i = 0;
while (i < length) {
sum = sum + array[i];
i = i + 1;
}
return sum;
}
```

- Mathematical Logic is the solution…..
  - Descriptive specification technique for specifying software…

Various Logic techniques

- Aristotelian logic
- Euclidean geometry
- Propositional logic
- First order logic
- Peano axioms
- Zermelo Fraenkel set theory
- Higher order logic

Propositional Logic

Propositional (Boolean) Logic

- In Propositional Logic (a.k.a Propositional Calculus or Sentential Logic), the objects are called propositions
- Definition
  - A proposition is a statement that is either true or false, but not both
- We usually denote a proposition by a letter: p, q, r, s, …

Introduction: Proposition

- The value of a proposition is called its truth value; denoted by
  - *T* or 1 if it is true or
  - *F* or 0 if it is false
- **Truth table**

p
0
1

Introduction: Proposition…

- The following are propositions
  - Today is Monday *M*
  - The grass is wet *W*
  - It is raining *R*
- The following are not propositions
  - C++ is the best language *Opinion*
  - When is the pretest? *Interrogative*
  - Do your homework *Imperative*

Logical connectives

- Connectives are used to create a compound proposition
  - Negation (denote ¬ or !)
  - And or logical conjunction (denoted $\wedge$)
  - Or or logical disjunction (denoted $\vee$)
  - XOR or exclusive or (denoted $\oplus$)
  - Implication (denoted $\Rightarrow$ or $\rightarrow$)
  - Biconditional (denoted $\Leftrightarrow$ or $\leftrightarrow$)

- We define the meaning (semantics) of the logical connectives using truth tables

Logical Connective: Implication

- Let p and q be two propositions. The implication $p \rightarrow q$ is the proposition that is false when p is true and q is false and true otherwise
  - p is called the hypothesis, antecedent, premise
  - q is called the conclusion, consequence
- Truth table

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Logical Connective: Implication…

- Examples
  - If you buy your air ticket in advance, it is cheaper.
  - If x is an integer, then $x^2 \geq 0$.
  - If 2+2=5, then all unicorns are pink.

Logical Connective: Biconditional

- The biconditional $p \leftrightarrow q$ is the proposition that is true when p and q have the same truth values. It is false otherwise.
- Note that it is equivalent to $(p \rightarrow q) \land (q \rightarrow p)$
- Truth table

p	q	$p \leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

Logical Connective: Biconditional…

- The biconditional $p \leftrightarrow q$ can be equivalently read as
  - p if and only if q
  - p is a necessary and sufficient condition for q
  - if p then q, and conversely

- Examples
  - The alarm goes off if and only if a burglar breaks in
  - "if I'm breathing, then I'm alive" and "if I'm alive, then I'm breathing"

Example : Informal statement

- A book can either be in stack, on reserve or loaned out.
- A book on loan can't be requested
- We want to,
  - Formalize the concept and statements
  - Prove the theorems to gain confidence that the spec. is correct.

Formalization

- Let’s first formalize some concepts:
  - S: the book is in the stack
  - R: the book is on reserve
  - L:the book is on loan
  - Q: the book can be requested

Formalization…

- A book can either be in stack, on reserve or loaned out
  - $S \land (\neg (R \lor L))$
  - $R \land (\neg (S \lor L))$
  - $L \land (\neg (S \lor R))$
- " A book on loan can't be requested "
  - $L \Rightarrow (\neg Q)$

Disadvantage of Propositional Logic

- Propositional logic has limited expressive power
  - unlike natural language
  - E.g., cannot say “Heavy snow-fall in Himalayas causes cold breeze in some regions of Gujarat “
    - except by writing one sentence for each region !!

First Order Logic

First Order Logic (FOL)

- Propositional logic assumes the world contains **facts.**
- First-order logic (like natural language) assumes the world contains
  - **Objects**: people, houses, wars, …
  - **Relations**: brother of, bigger than, part of, comes between, …
  - **Functions**: one more than, plus, power set …

Syntax of FOL: Basic elements

- Constant Symbols:
  - Stand for objects
  - e.g., Abdul Kalam, 2, NIT,...
- Predicate Symbols
  - Stand for relations
  - E.g., Brother(Ram, Bharat), greater_than(3,2)...
- Function Symbols
  - Stand for functions
  - E.g., Sqrt(3), mul(x,y),...

Syntax of FOL: Basic elements…

- Constants Abdul Kalam, 2, NIT, ...
- Predicates Brother, >, ...
- Functions Sqrt, mul, ...
- Variables x, y, a, b, ...
- Connectives $\neg, \Rightarrow, \wedge, \vee, \Leftrightarrow$
- Equality =
- Quantifiers $\forall, \exists$

First Order Logic in Software Specification

Check validity of address

Example – Saving addresses

// name must not be empty

// state must be valid

// zip must be 5 numeric digits

// street must not be empty

// city must not be empty

Rewriting to logical expression

name != “” ∧ state in stateList ∧ zip >= 00000 ∧ zip <= 99999 ∧ street != “” ∧ city != “”

Specifying complete programs: Hoare Triple

A *property*, or *requirement*, for P is specified as a formula of the type

$$\{Pre\ (i_1, i_2, \ldots, i_n)\ \}$$
$$P$$
$$\{Post\ (o_1, o_2, \ldots, o_m, i_1, i_2, \ldots, i_n)\}$$

Pre: precondition
Post: postcondition

Specifying complete programs

- PRE: FOT formula having $i_1, i_2, \ldots, i_n$ as free variables
- POST: FOT formula having $o_1, o_2, \ldots, o_m$, and possibly $i_1, i_2, \ldots, i_n$ as free variables
- PRE :Precondition of P
- POST :Post condition of P
- *The preceding formula is intended to mean that if PRE holds for the given input values before P's execution, then, after P finishes executing, POST must hold for the output and input values*

Examples

- Simple requirement of the division

$$\{\text{exists } z\ (i_1 = z * i_2)\}$$

$$P$$

$$\{o_1 = i_1/i_2\}$$

Examples…

- Stronger requirement of the division

$$\{i_1 > i_2\}$$
$$P$$
$$\{i_1 = i_2 * o_1 + o_2 \textbf{ and } o_2 >= 0 \textbf{ and } o_2 < i_2\}$$

- Imposes more constraints on output values less on input values
- A precondition {true} does not place any constraint on input values

Examples…

- Requires that P produce greater of i_1 and i_2

  {true}
  P
  {($o = i_1$ **and** $o >= i_2$)|| ($o = i_2$ **and** $o>=i_1$)}

- Program to compute sum of the input sequence

  {n > 0}
  P
  $$\{ O = \sum_{k=1}^{n} i_k \}$$

Algebraic specifications

- For formally specifying system behavior.
- Formally define types of data and mathematical operations on those data types.
- Abstracting implementation details, such as the size of representations (in memory) and the efficiency of obtaining outcome of computations.

Example

- A system for strings, with operations for
  - creating new, empty strings (operation new)
  - concatenating strings (operation append)
  - adding a new character at the end of a string (operation add)
  - checking the length of a given string (operation length)
  - checking whether a string is empty (operation isEmpty)
  - checking whether two strings are equal (operation equal)

Specification: syntax

algebra StringSpec;
introduces
    sorts String, Char, Nat, Bool;
    operations
        new: () → String;
        append: String, String → String;
        add: String, Char → String;
        length: String → Nat;
        isEmpty: String → Bool;
        equal: String, String → Bool

Specification: properties

```
constrains new, append, add, length, isEmpty, equal so that
for all [s, s1, s2: String; c: Char]
    isEmpty (new ()) = true;
    isEmpty (add (s, c)) = false;
    length (new ()) = 0;
    length (add (s, c)) = length (s) + 1;
    append (s, new ()) = s;
    append (s1, add (s2,c)) = add (append (s1,s2),c);
    equal (new (),new ()) = true;
    equal (new (), add (s, c)) = false;
    equal (add (s, c), new ()) = false;
    equal (add (s1, c), add (s2, c) = equal (s1,s2);
end StringSpec.
```

Example: editor

- newF
  - creates a new, empty file
- isEmptyF
  - states whether a file is empty
- addF
  - adds a string of characters to the end of a file
- insertF
  - inserts a string at a given position of a file (the rest of the file will be rewritten just after the inserted string)
- appendF
  - concatenates two files

algebra TextEditor;
introduces
sorts Text, String, Char, Bool, Nat;
operations
newF: () → Text;
isEmptyF: Text → Bool;
addF: Text, String → Text;
insertF: Text, Nat, String → Text;
appendF: Text, Text → Text;
deleteF: Text → Text;
lengthF : Text → Nat;
equalF : Text, Text → Bool;
addFC: Text, Char → Text;
{This is an auxiliary operation that will be needed to define addF and other operations on files.}

```
constrains newF, isEmptyF, addF, appendF, insertF, deleteF
so that TextEditor generated by [newF, addFC]
for all [f, f1,f2: Text; s: String; c: Char; cursor: Nat]
    isEmptyF (newF ()) = true;
    isEmptyF (addFC (f, c)) = false;
    addF (f, newS ()) = f;
    addF (f, addS (s, c)) = addFC (addF (f, s), c);
    lengthF (newF ()) = 0;
    lengthF (addFC (f, c)) = lengthF (f) + 1;
    appendF (f, newF ()) = f;
    appendF (f1, addFC (f2, c)) =
        addFC (appendF (f1, f2), c);
    equalF (newF (),newF ()) = true;
    equalF (newF (), addFC (f, c)) = false;
    equalF (addFC (f, c), new ()) = false;
    equalF (addFC (f1, c1), addFC (f2, c2) =
    equalF (f1, f2) and equalC (c1, c2);
    insertF (f, cursor, newS ()) = f;
    end TextEditor.
```

Incremental specification of an ADT

- We want to target stacks, queues, sets
- We start from "container" and then progressively specialize it
- We introduce another structuring clause
  - assumes
    - defines inheritance relation among algebras

Container algebra

```
algebra Container;
imports DataType, BoolAlg, NatNumb;
introduces
    sorts Cont;
    operations
        new: () → Cont;
        insert: Cont, Data → Cont;
            {Data is the sort of algebra DataType, to which
            elements to be stored in Cont belong}
        isEmpty: Cont → Bool;
        size: Cont → Nat;
constrains new, insert, isEmpty, size so that
Cont generated by [new, insert]
for all [d: Data; c: Cont]
    isEmpty (new ()) = true;
    isEmpty (insert (c, d)) = false;
    size (new ()) = 0;
end Container.
```

Queue specializes Container

```
algebra QueueContainer;
assumes Container;
introduces
    sorts Queue;
    operations
        last: Queue → Data;
        first: Queue → Data;
        equalQ : Queue , Queue → Bool;
        delete:Queue → Queue;
constrains last, first, equalQ, delete, isEmpty, new, insert so that
for all [d: Data; q, q1, q2: Queue]
    last (insert (q, d)) = d;
    first (insert (new(), d) = d
    first (insert (q, d)) = if not isEmpty (q) then first (q);
    equalQ (new (), new ()) = true;
    equalQ (insert (q, d), new ()) = false;
    equalQ (new (), insert (q, d)) = false;
    equalQ (insert (q1, d1), insert (q2, d2)) = equalD (d1, d2) and
    equalQ (q1,q2);
    delete (new ()) = new ();
    delete (insert (new (), d)) = new ();
end QueueContainer.
```

From specs to an implementation

- Algebraic spec language described so far is based on the "Larch shared language"
- Several "interface languages" are available to help transitioning to an implementation
  - Larch/C++, Larch/Pascal

Languages for modular specifications

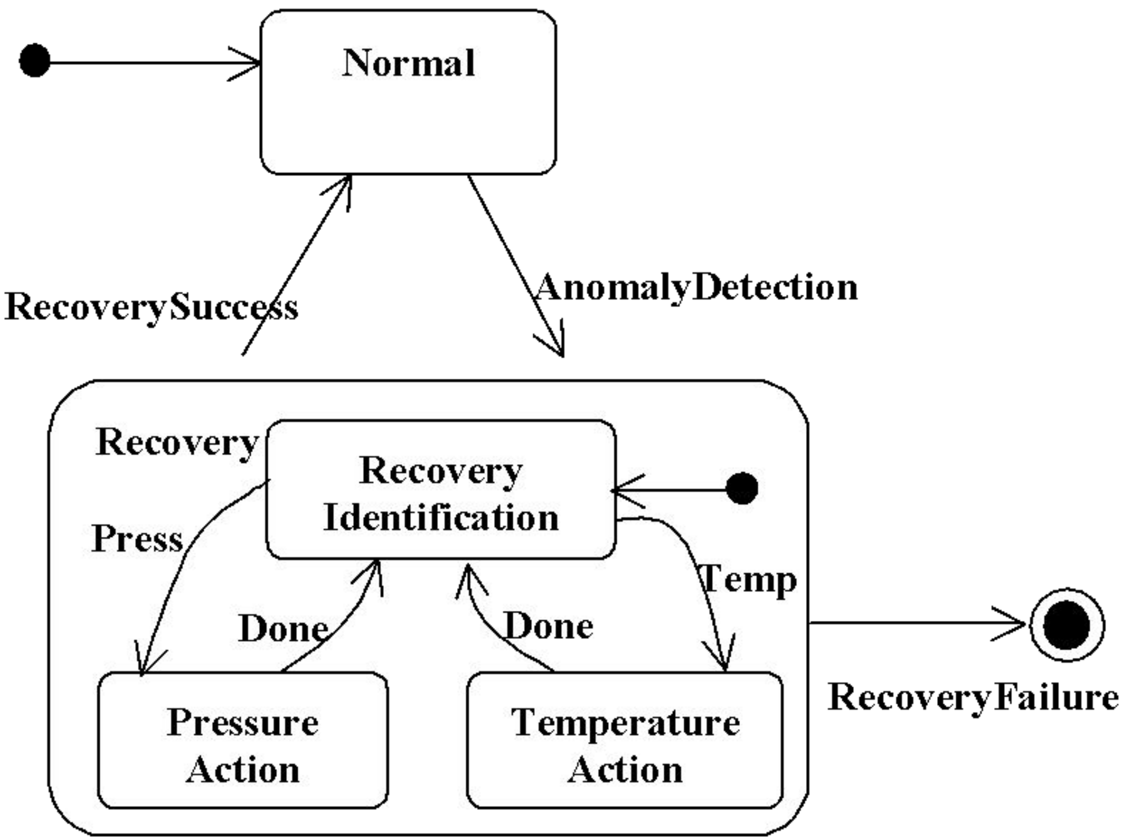
Statecharts

Z

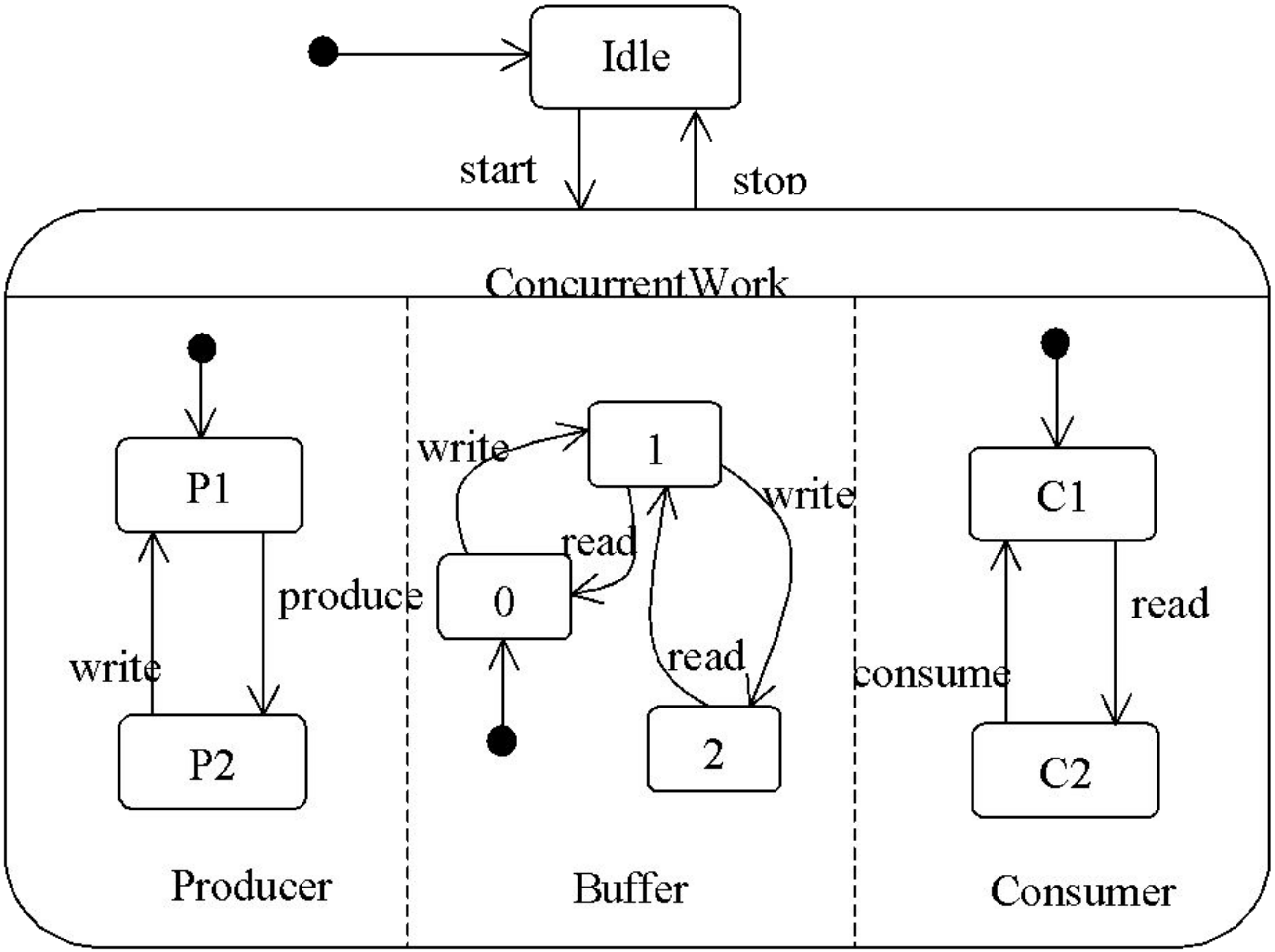
Modularizing finite state machines

- Statecharts do that
- They have been incorporated in UML
- They provide the notions of
  - superstate
  - state decomposition

Sequential decomposition --chemical plant control example--



Parallel decomposition



Modularizing logic specifications: Z

- System specified by describing state space, using *Z schemas*
- Properties of state space described by *invariant* predicates
  - predicates written in first-order logic
- Operations define state transformations

The elevator example in Z

$SWITCH ::= on \mid off$

$MOVE ::= up \mid down$

$$
\begin{array}{|l}
FLOORS : \mathbb{N} \\
\hline
FLOORS > 0
\end{array}
$$

IntButtons

$IntReq : 1 \,..\, FLOORS \rightarrow SWITCH$

FloorButtons

$ExtReq : 1 \,..\, FLOORS \rightarrow \mathbb{P}\, MOVE$

$down \notin ExtReq(1)$

$up \notin ExtReq(FLOORS)$

Scheduler

$NextFloorToServe : 0 \,..\, FLOORS$

Elevator

$CurFloor : 1 \,..\, FLOORS$

$CurDirection : MOVE$

Complete state space attempt #1

$$
\begin{array}{|l}
\hline
\textit{System} \\
\textit{Elevator} \\
\textit{IntButtons} \\
\textit{FloorButtons} \\
\textit{Scheduler} \\
\hline
\textit{NextFloorToServe} \neq 0 \\
\Rightarrow \textit{IntReq}(\textit{NextFloorToServe}) = \textit{on} \lor \textit{ExtReq}(\textit{NextFloorToServe}) \neq \emptyset \\
\hline
\end{array}
$$

Complete state space attempt #2

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{System}} \\
\hline
\textit{Elevator} \\
\textit{IntButtons} \\
\textit{FloorButtons} \\
\textit{Scheduler} \\
\hline
\textit{NextFloorToServe} \neq 0 \Rightarrow \\
\qquad \textit{IntReq}(\textit{NextFloorToServe}) = \textit{on} \lor \textit{ExtReq}(\textit{NextFloorToServe}) \neq \emptyset \\
\textit{NextFloorToServe} = 0 \Rightarrow \\
\qquad (\forall f : 1 \mathinner{..} \textit{FLOORS} \bullet (\textit{IntReq}(f) = \textit{off} \land \textit{ExtReq}(f) = \emptyset)) \\
\hline
\end{array}
$$

Complete state space final

System

Elevator
IntButtons
FloorButtons
Scheduler

$\exists\ Pri1, Pri2, Pri3 : \mathbb{P}\,\mathbb{N}_1 \bullet$
$CurDirection = up \Rightarrow$
$(Pri1 = \{f : 1\,..\,FLOORS \mid f \geq CurFloor \wedge (IntReq(f) = on \vee up \in ExtReq(f))\} \wedge$
$Pri2 = \{f : 1\,..\,FLOORS \mid down \in ExtReq(f) \vee (f < CurFloor \wedge IntReq(f) = on)\} \wedge$
$Pri3 = \{f : 1\,..\,FLOORS \mid f < CurFloor \wedge up \in ExtReq(f)\} \wedge$
$((Pri1 \neq \emptyset \wedge NextFloorToServe = min(Pri1)) \vee$
$(Pri1 = \emptyset \wedge Pri2 \neq \emptyset \wedge NextFloorToServe = max(Pri2)) \vee$
$(Pri1 = \emptyset \wedge Pri2 = \emptyset \wedge Pri3 \neq \emptyset \wedge NextFloorToServe = min(Pri3))$
$\vee (Pri1 = \emptyset \wedge Pri2 = \emptyset \wedge Pri3 = \emptyset \wedge NextFloorToServe = 0))) \wedge$
$CurDirection = down \Rightarrow$
$(Pri1 = \{f : 1\,..\,FLOORS \mid f \leq CurFloor \wedge$
$(IntReq(f) = on \vee down \in ExtReq(f))\} \wedge$
$Pri2 = \{f : 1\,..\,FLOORS \mid up \in ExtReq(f) \vee$
$(f > CurFloor \wedge IntReq(f) = on)\} \wedge$
$Pri3 = \{f : 1\,..\,FLOORS \mid f > CurFloor \wedge down \in ExtReq(f)\} \wedge$
$((Pri1 \neq \emptyset \wedge NextFloorToServe = max(Pri1)) \vee$
$(Pri1 = \emptyset \wedge Pri2 \neq \emptyset \wedge NextFloorToServe = min(Pri2)) \vee$
$(Pri1 = \emptyset \wedge Pri2 = \emptyset \wedge Pri3 \neq \emptyset \wedge NextFloorToServe = max(Pri3))$
$\vee (Pri1 = \emptyset \wedge Pri2 = \emptyset \wedge Pri3 = \emptyset \wedge NextFloorToServe = 0)))$

Operations (1)

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MoveToNextFloor}} \\
\Delta System \\
\hline
NextFloorToServe \neq 0 \\
CurFloor \neq NextFloorToServe \\
CurFloor > NextFloorToServe \Rightarrow \\
CurFloor' = CurFloor - 1 \land CurDirection' = down \\
CurFloor < NextFloorToServe \Rightarrow \\
CurFloor' = CurFloor + 1 \land CurDirection' = up \\
\theta IntButtons' = \theta IntButtons \\
\theta FloorButtons' = \theta FloorButtons \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InternalPush}} \\
\Delta System \\
f? : 1 \ldots FLOORS \\
\hline
IntReq' = IntReq \oplus \{f? \mapsto on\} \\
\theta Elevator' = \theta Elevator \\
\theta FloorButtons' = \theta FloorButtons \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ExternalPush}} \\
\Delta System \\
f? : 1 \ldots FLOORS \\
dir? : MOVE \\
\hline
ExtReq' = ExtReq \oplus \{(f? \mapsto (ExtReq(f?) \cup \{dir?\}))\} \\
\theta Elevator' = \theta Elevator \\
\theta IntButtons' = \theta IntButtons \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ServeIntRequest}} \\
\Delta System \\
\hline
NextFloorToServe = CurFloor \\
IntReq(CurFloor) = on \\
IntReq' = IntReq \oplus \{(CurFloor \mapsto off)\} \\
ExtReq' = ExtReq \\
CurFloor' = CurFloor \\
CurDirection' = CurDirection \\
\hline
\end{array}
$$

Operations (2)

$$
\begin{array}{l}
\textit{ServeExtRequestSameDir} \\
\hline
\Delta System \\
\hline
NextFloorToServe = CurFloor \\
IntReq(CurFloor) = \mathit{off} \\
CurDirection \in ExtReq(CurFloor) \\
IntReq' = IntReq \\
ExtReq' = ExtReq \oplus \{(CurFloor \mapsto (ExtReq(CurFloor) \setminus \{CurDirection\}))\} \\
CurFloor' = CurFloor \\
CurDirection' = CurDirection \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{ServeExtRequestOtherDir} \\
\hline
\Delta System \\
\hline
NextFloorToServe = CurFloor \\
IntReq(CurFloor) = \mathit{off} \\
CurDirection \notin ExtReq(CurFloor) \\
IntReq' = IntReq \\
ExtReq' = ExtReq \oplus \{(CurFloor \mapsto \emptyset)\} \\
CurFloor' = CurFloor \\
CurDirection' = CurDirection \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{SystemInit} \\
\hline
System' \\
\hline
\forall\, i : 1..FLOORS \bullet IntReq'(i) = \mathit{off} \wedge ExtReq'(i) = \emptyset \\
NextFloorToServe' = 0 \\
CurFloor' = 1 \\
CurDirection' = up \\
\hline
\end{array}
$$

Conclusions (1)

- Specifications describe
  - what the users need from a system (requirements specification)
  - the design of a software system (design and architecture specification)
  - the features offered by a system (functional specification)
  - the performance characteristics of a system (performance specification)
  - the external behavior of a module (module interface specification)
  - the internal structure of a module (internal structural specification)

Conclusions (2)

- Descriptions are given via suitable notations
  - There is no “ideal” notation
- They must be modular
- They support communication and interaction between designers and users