CSI 5109

Specification Methods for Distributed Systems

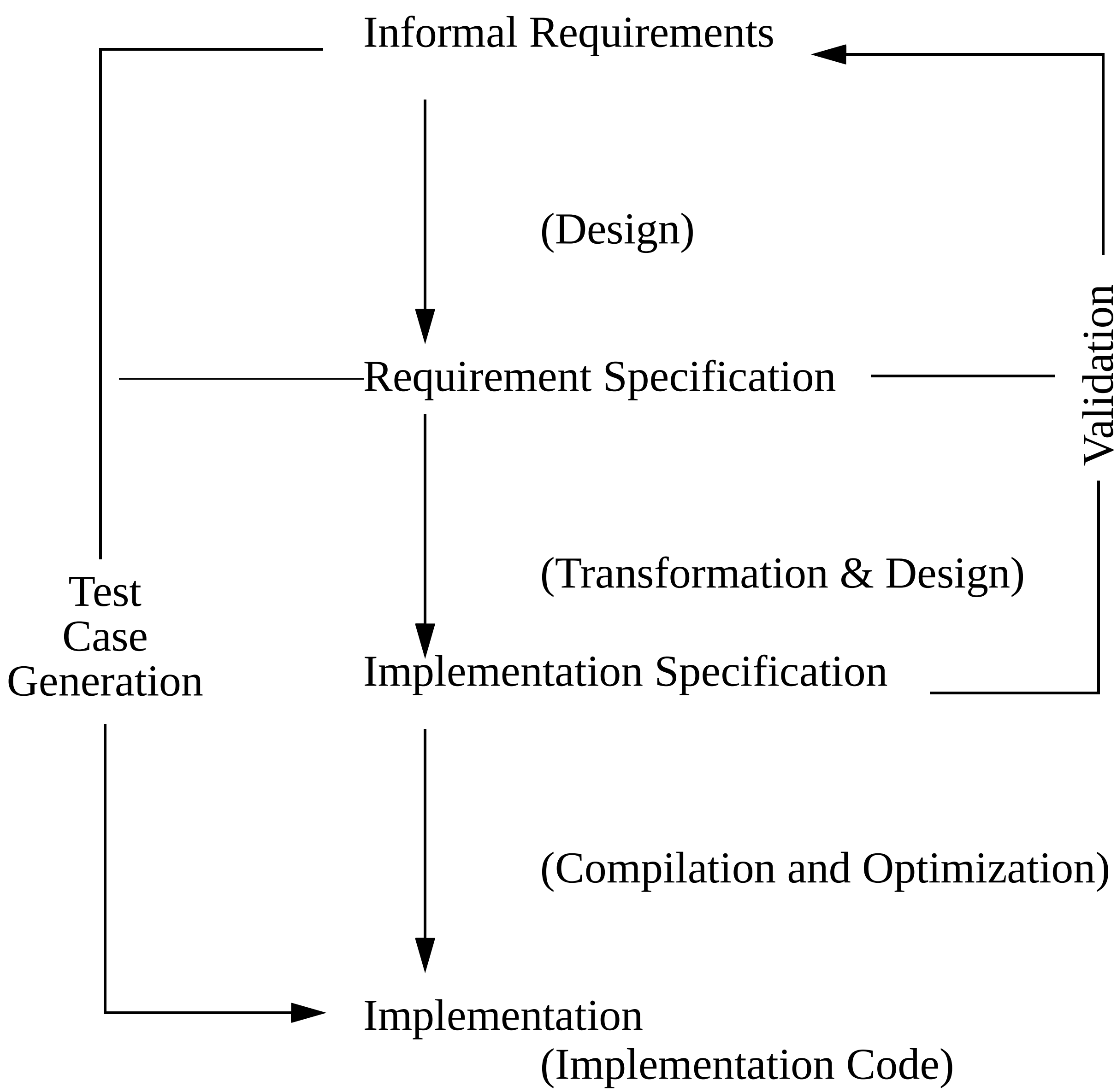
Course notes by

Luigi Logrippo
and the LOTOS group

University of Ottawa
School of Information Technology and Engineering
Telecommunications Software Engineering Research Group

please read course description reachable from prof's homepage:
www.site.uottawa.ca/~luigi

A (Simplistic) View of the Software Development Process



The earlier a potential problem
is detected in the software lifecycle,
the smaller its cost is.

**By specifying and validating a system
at the design stage,
many design ambiguities and errors
can be identified
before implementation starts**

Formal Description Techniques (FDTs) or Formal Specification Languages

Languages for the precise specification of the functionalities of software.

They must have a math basis (*formal*).

They can be used in all phases of the process: requirements, implementation, testing...

In what do they differ from programming languages?

At the specification level, there is no need for execution efficiency, so we can use more abstract concepts

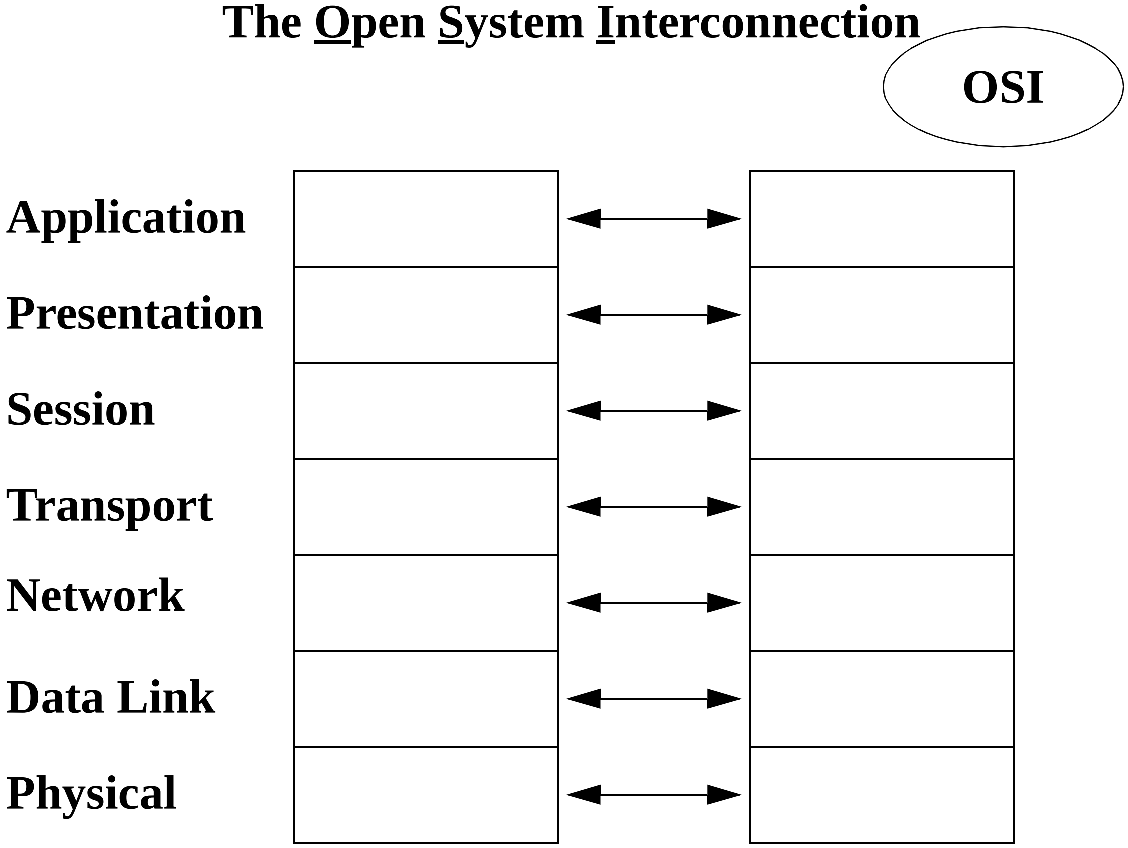
However:

> Is today’s specification language tomorrow’s programming language?

HISTORY

In 1978, ISO and CCITT (now ITU) set out to define
a new family of standards for data communications protocols

The Open System Interconnection



The Seven-Layer "Reference Model"

OSI was never fully implemented,
however it generated many useful ideas

FDT for OSI

One of the requirements of OSI standardization was to have one (or more!) techniques for the **formally precise** specification of the standards.

(A standard can only be effective if it is precisely specified, in an implementation-independent fashion)

Note: in this course, we shall be concerned only with telecom-related FDTs.

FDTs

USE	FEATURE
Directives to implementors, Communication between implementors. Standardization	Well-understood, unambiguous, precise, simple All the "natural" concepts, abstract, non-procedural
Verification, formal testing	Logically sound -> Proofs Executable, Support of test techniques
Design tools	Modularity, abstraction levels

A Great Number of Open Problems:

Expressive Languages

Suitable both for automatic processing and human use to express requirements at various levels of abstraction.

Design and Transformation Techniques

To move between levels of abstraction

Validation Techniques

To prove consistency between levels of abstraction

Derivation of Test Cases

To test consistency

Etc. Etc.

Theoretical Developments

Happily, just as these needs were being recognized, some theories were being developed, which had potential for meeting them.

- Automata theory and its extensions
- Language theory and its extensions
- Theory of Communicating Sequential Processes (Hoare’s CSP)
- Calculus of Communicating Systems (Milner’s CCS)
- Temporal Logic

etc.

Some standard FDTs

SDL (Specification and Description Language)	Extended FSM model	An ITU (CCITT) standard
ESTELLE	Extended FSM model - Pascal-base	An ISO standard
LOTOS	CCS, CSP, Act One	An ISO standard
ASN.1	A notation for specifying data formats	An ISO/ITU standard
TTCN (Tree and Tabular Combined Notat.)	A notation for specifying test cases	An ISO/ITU standard

A Personal View of FDT progression

In the early seventies, the main model for telecom specification was the *communicating finite-state machine* (FSM) model. This was the initial SDL model.

This model was not powerful enough to express protocols (little data), hence the Extended FSM model: today's SDL, Estelle.

Process algebras (CCS, CSP) came about in the eighties. They had more sophisticated semantics, expressed in LOTOS.

E-LOTOS (Extended LOTOS), defined in the nineties, adds extra expressive power but the basic model is the same.

We can expect more sophisticated models from newer theoretical developments (e.g. category theory, linear logic). However efforts in this direction are still immature.

LANGUAGE

OF

TEMPORAL

ORDERING

SPECIFICATIONS

Meaning that the language defines the temporal ordering of events in time (but not their precise time of occurrence).

Consists of *two main parts*:

Behavior part, to specify ordering of events. Main influences:

- CCS: Milner’s Calculus of Communicating Systems
- CSP: Hoare’s Communicating Sequential Processes

Data part, to specify the data abstractions:

- ACT ONE Abstract Data Type formalism

Ohm’s Law

V / A = R

or

V = A X R

or

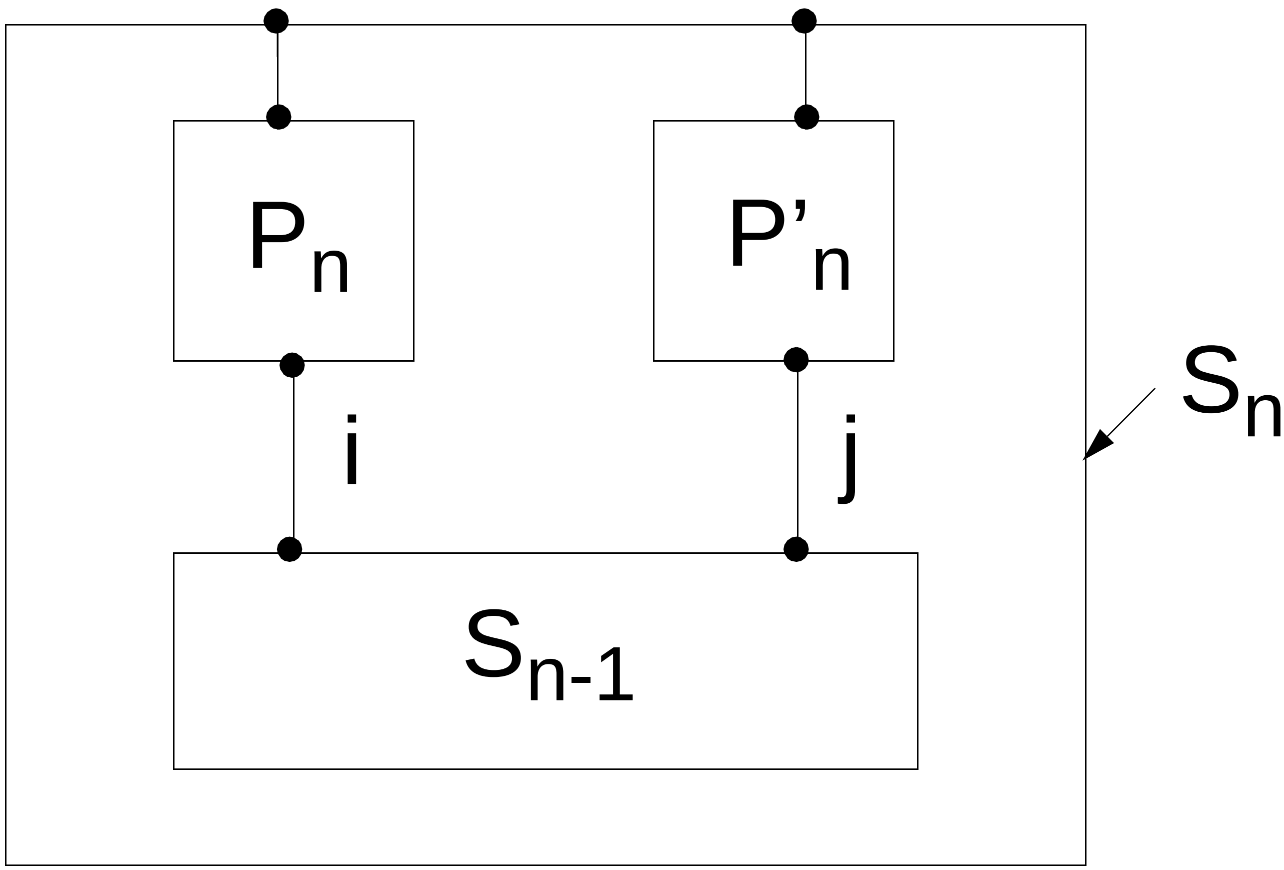
A = V / R

**Engineering is made possible largely
by the existence of such laws,
and by our ability to manipulate them
symbolically, yielding other useful laws.
*This is algebra***

Formal methods
in software seek to use mathematics in software in the same way as mathematics have been used in engineering: to provide symbolic models for the behaviour and properties of an artifact.

What we need is a *mathematics of distributed software* by which programs and specifications can be manipulated symbolically with the same ease as we can manipulate algebraic expressions.

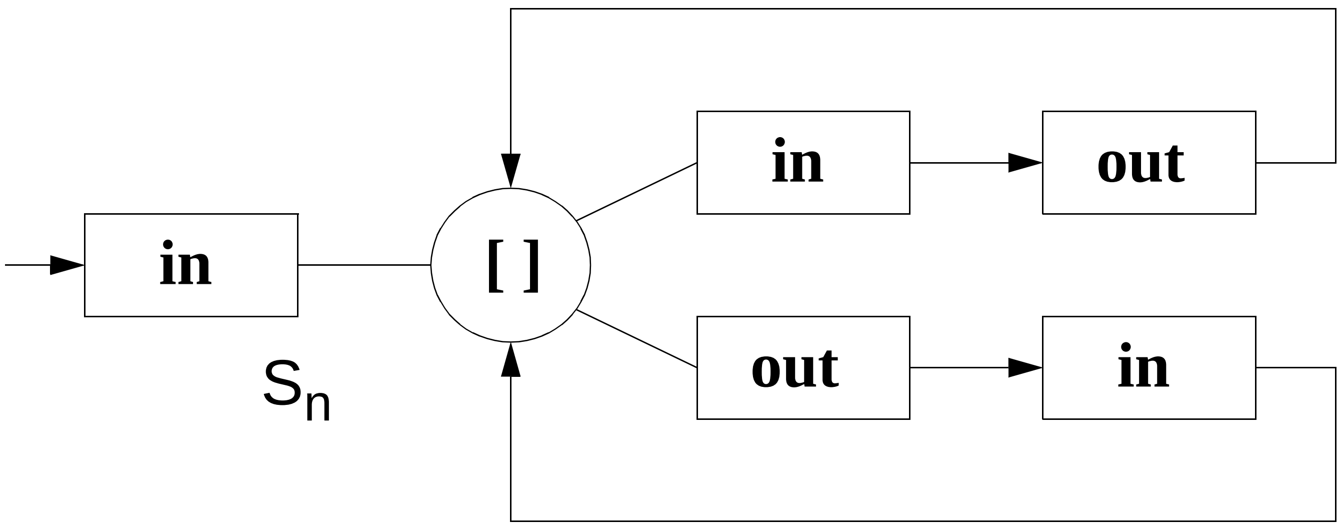
E.g., in a layered protocol model (OSI-like)

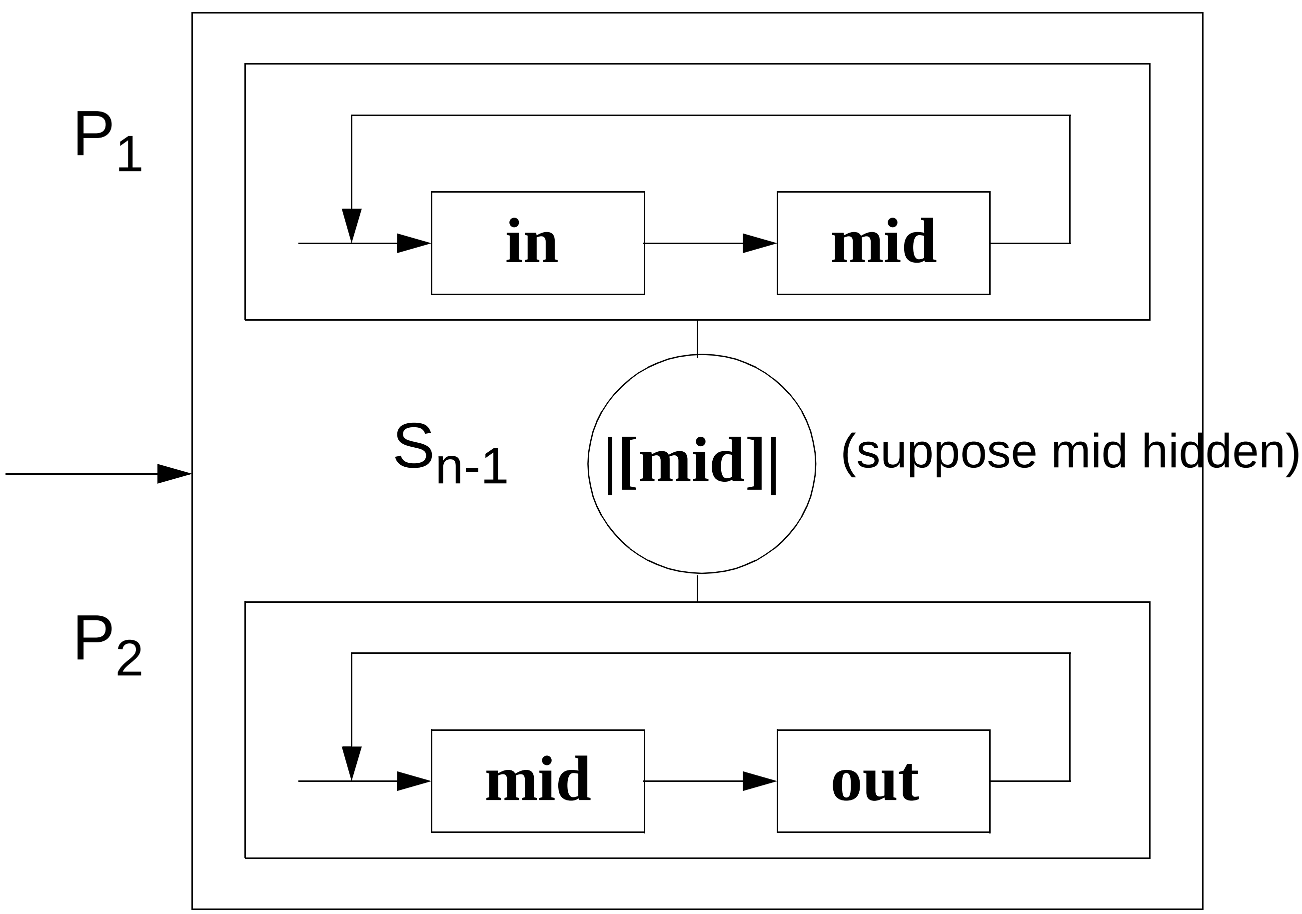


it would be useful to be able to conclude formally that $S_n = (P_n ||| P'_n) | [i,j] | S_{n-1}$

(let S_n be a service provider of layer S, which is realized by two protocols P_n and P'_n over a service provider of layer S_{n-1}. These communicate with S by two service access points i and j)

An example of very simple equivalence provable in LOTOS:





Two different specifications for a buffer capable of storing a maximum of two signals.

A real *mathematics of distributed softwar*e does not yet exist. However, its laws are slowly becoming understood (how many centuries after the first bridge did the engineering and mathematical laws underlying bridge construction become clear?)

The need of *efficient computation* on a Von Neumann machine leads to sequential thought which is not conducive to mathematical thinking. Many small functions are involved, the order of execution is important, functions have side effects.

The logical clumsiness of low-level software can be hidden by increasingly abstract languages, that have increasingly nicer mathematical properties. plus rules of transformations between such languages.

LOTOS is a specification language for distributed systems where many "nice" mathematical properties (a *process algebra*) hold:

$$\mathbf{B_1||B_2 = B_2||B_1}$$
$$\mathbf{B_1[]B_2 = B_2[]B_1}$$

$$\mathbf{B_1 \; || \; (B_2 \; || \; B_2) = (B_1 \; || \; B_2) \; || \; B_3}$$

$$\mathbf{B_1 \; [] \; (B_2 \; [] \; B_3) = (B_1 \; [] \; B_2) \; [] \; B_3}$$

$$\mathbf{B \; || \; stop = stop}$$
$$\mathbf{B \; [] \; stop = B}$$

etc., etc.

It is possible to manipulate LOTOS formulae according to these algebraic identities, yielding conclusions about equivalence of formulae, etc..

This course will concentrate on LOTOS.

However it intends to be a general introduction on

algebraic techniques

for specifying, validating and testing distributed systems and their protocols

LOTOS PHILOSOPHY

Executability:

- **Fast Prototyping**
  → early detection of design errors

- **Lifecycle Support**

Mathematical Basis:

- **Algebraic manipulation of Specifications**

- **Support of various types of Validation**

Some basic LOTOS ideas (will be clearer later):

Formal, mathematical definition:

Syntax (usual BNF)

Static Semantics (attributed grammar)

Dynamic Operational Semantics
(inference rules)

Data algebra for defining data properties

Process algebra for defining behavior properties

Process encapsulation [object *based*]
processes communicating by messages

Process parameterization

Interleaving parallelism:

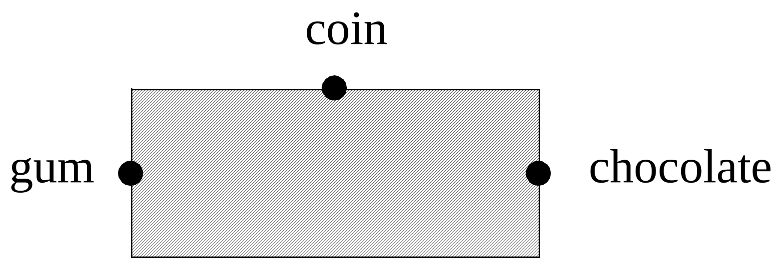
A in *parallel* with B means that actions of A and B are allowed to *interleave* arbitrarily

LOTOS AND SOFTWARE LIFECYCLE

	FEATURES	TOOLS
DESIGN	Expressiveness Modularity	Graphic LOTOS-Graphic editors Syntax, static semantics checkers
DESIGN VERIFICATION	Precisely defined semantics Verification rules Executability of Specifications	Computer-assisted verification tools for static, dynamic verification
IMPLEMENTATION	Executable	Computer-assisted translation "abstract_LOTOS" -> implementation -> executable code
VERIFICATION AND TESTING OF IMPLEMENTATION	Precise testing theory Test trees can be obtained from specifications Verification rules	Generation of "useful" execution trees and test sequences. Computer-assisted verification tools
MAINTENANCE	Documentation Modularity Implementation independence	Verification of "persistence" of properties

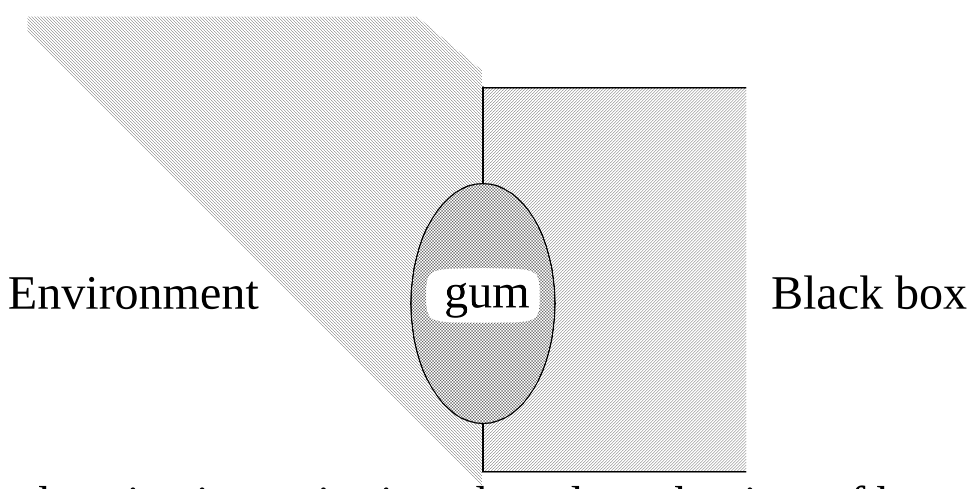
A “Black Box” and its Interaction Points

(or “synchronization points”)



Execution of an “action” at a “gate” requires participation of both environment and box

So the execution of an action is called “synchronization”



The synchronization point is a shared mechanism of box and environment

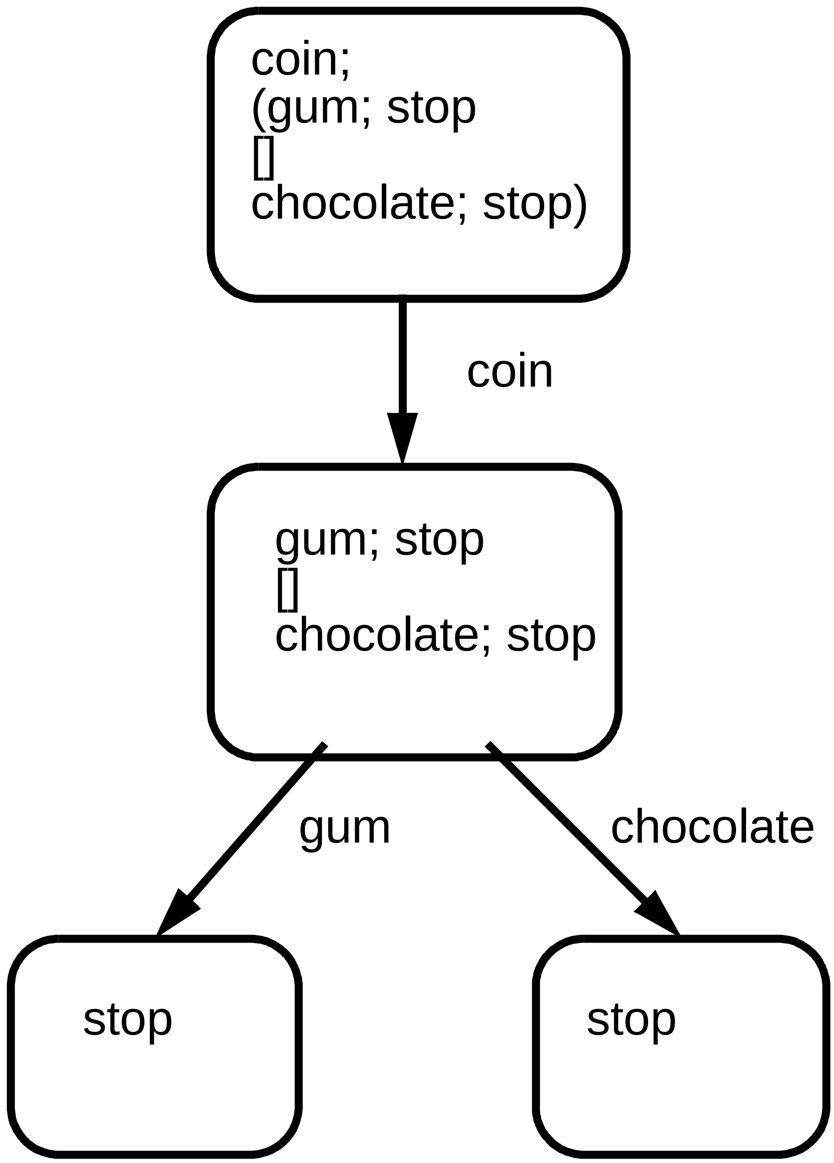
Environment can be other processes or the external world

Some synonyms: interaction, synchronization, rendezvous

A LOTOS *behavior expression*:

```
coin;
   (gum; stop
   []
    chocolate; stop)
```

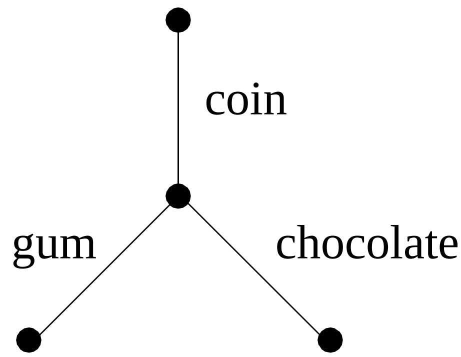
describes a black box that can take a *coin* and then give *gum* or *chocolate*.



A LOTOS behavior tree: states are
behavior expressions, transitions are actions.
(also called: synchronization tree,
Labeled Transition System...)

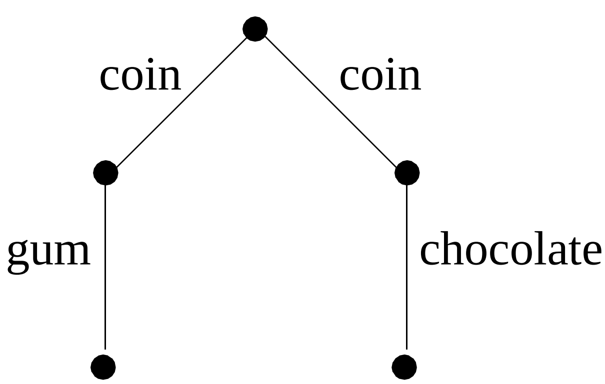
Behavior Trees to Represent Finite Behaviors

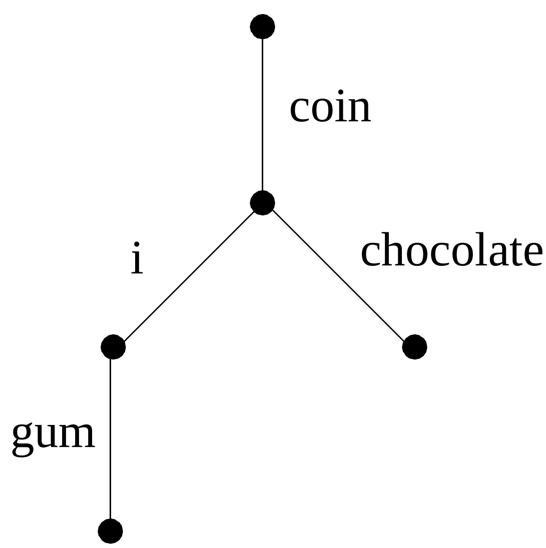
(Also called *synchronization trees*)



What happens if one tries
to select gum before coin?

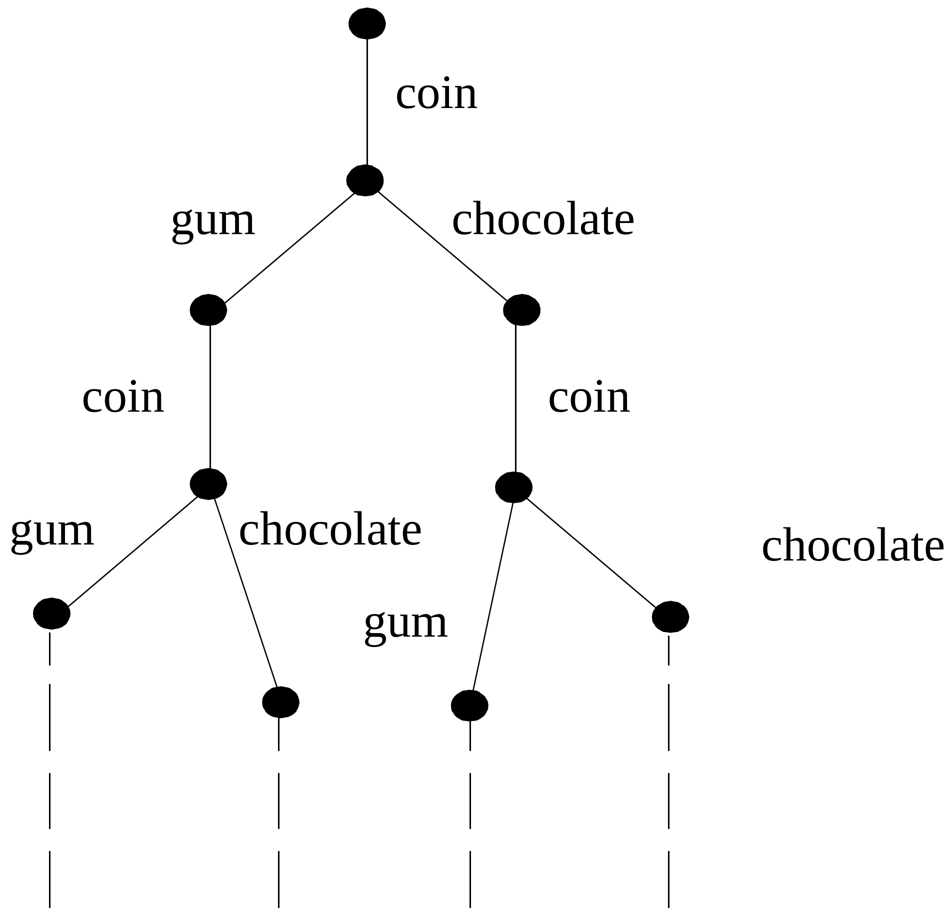
What happens if one tries
to put a second coin in?



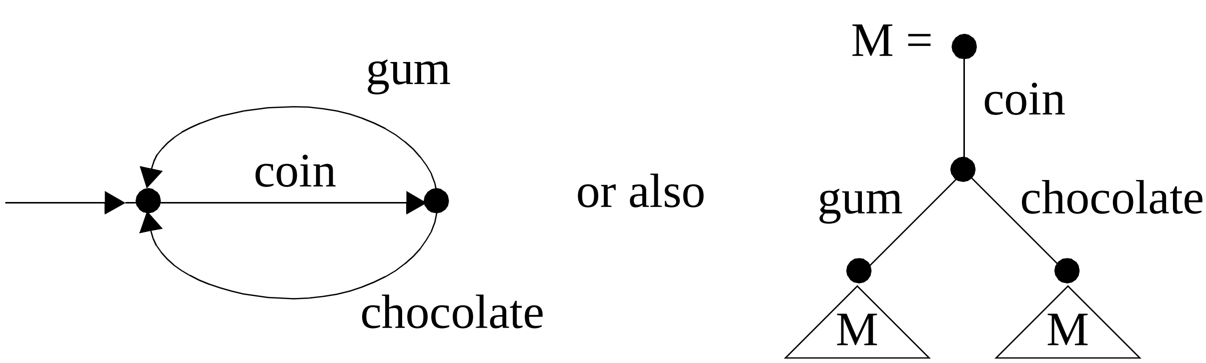


Any difference in observable behavior?

Trying to represent infinite behavior
by behavior trees



A *labeled transition system*



A Labelled Transition System (LTS) is a directed, connected graph where each edge (denoting a state transition) is labelled by the name of an action,

- external or
- internal (= **i**).

LTSs are used to describe the behavior of processes.
Traversal of an edge labelled by an external action means that the environment and the process have agreed to **execute** the action.

We also say that the environment and the process **synchronize** on the action.However **i** can be executed independently by a process.

LTSs are also called **behavior** or **synchronization** graphs. When *unfolded* as trees, they are called *behavior* or *synchronization trees*.

Note:
Process instantiations, guards, behavior expressions such as **stop** or **exit** etc. cannot label edges.

Behavior Expressions in LOTOS

A behavior expression represents at the same time

• the state of a process (i.e. a black box, a system)
• its labelled transition system

In other words, the state of a process is the set
of potential behaviors of the process.

There are two predefined behavior expressions:

stop or deadlock
exit or successful termination

There is one predefined action:

i the internal action

More complex behavior expressions can be obtained by combining actions and behavior expressions by using operators, process definitions, process instantiations, etc. as we shall see ...

A LOTOS behavior expression specifies the "externally observable" behavior of processes which communicate with the environment by means of “actions” at "gates":

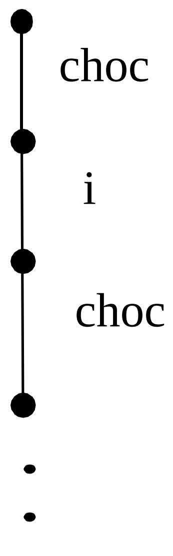
E.g.

```
    process P [choc] :=
        choc; hide rumble in rumble; P[choc] (*recursion!*)
    endproc
```

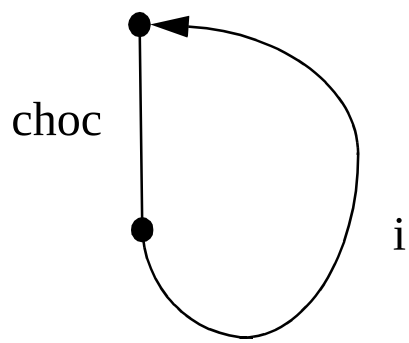
is a process that keeps offering to interact with the environment at gate *choc*. It hides action *rumble*. Recursion is used to represent a repeating process.

if the environment collaborates,
  P will keep offering choc
if at some point the environment does not collaborate,
  P will **stop** (or "deadlock").

Behavior tree for P:



LTS for P:



Basic LOTOS Operators

; Action prefix (Sequencing actions and behavior)

[ ] Choice

||

| [...] | Parallel composition

|||

hide Hide some actions

>> Enabling (Sequencing two behaviors)

[> Disabling (A behavior can interrupt another behavior)

P (...) [...] Process Instantiation

Concepts discussed in Class 1:

- Why formal techniques
- Some history
- Basic principles of LOTOS:
  - algebraic
  - concept of synchronization (symmetric)
  - environment
  - states and transitions
  - external and internal actions
  - behavior trees, LTS
  - behavior expressions as states
  - list of basic operators

And whosoever of them ate the honey-sweet fruit of the lotos [...] wanted to abide there among the lotos-eaters, feeding on the lotos

(Homer, Odyssey)

Reference Materials to get started

Go to:

http://LOTOS.csi.UOttawa.ca/ftp/pub/Lotos/Intro/

Download

Logrippo, L., Faci, M., and Haj-Hussein, M. "An Introduction to LOTOS: Learning by Examples.

Bolognesi, T., and Brinksma, E. Introduction to the ISO Specification Language LOTOS

Turner Tutorial

At the Reserve in the Library, look for

K.J. Turner (ED.) Using Formal Description Techniques (presents in detail LOTOS, Estelle, SDL with many examples)

Another introduction to LOTOS written by Ken Turner

Other materials on reserve (useful for projects and for those who want to know a lot...):

Books by Milner (on CCS) and Hoare (on CSP), and by Ehrig and Mahr (on ACT-ONE)

Two books on European projects related to LOTOS

The ISO LOTOS standard