Semantic Formalisms 1:
An Overview

- Formal Methods
  Operational Semantics
  CCS, Equivalences
- Software Components
  Fractal : hierarchical components
  Deployment, transformations
  Specification of components
- Application to distributed applications
  Active object and distributed components
  Behaviour models
  An analysis and verification platform

Eric Madelaine
eric.madelaine@sophia.inria.fr

INRIA Sophia-Antipolis
Oasis team

UNICE – EdStic
Mastère Réseaux et Systèmes Distribués
TC4

www-sop.inria.fr/oasis/Eric.Madelaine/Teaching/

Program of the course: 1: Semantic Formalisms

- Semantics and formal methods:
  - motivations, definitions, examples
- Operational semantics, behaviour models : represent the complete behaviour of the system
  - CCS, Labelled Transition Systems
  - Equivalences

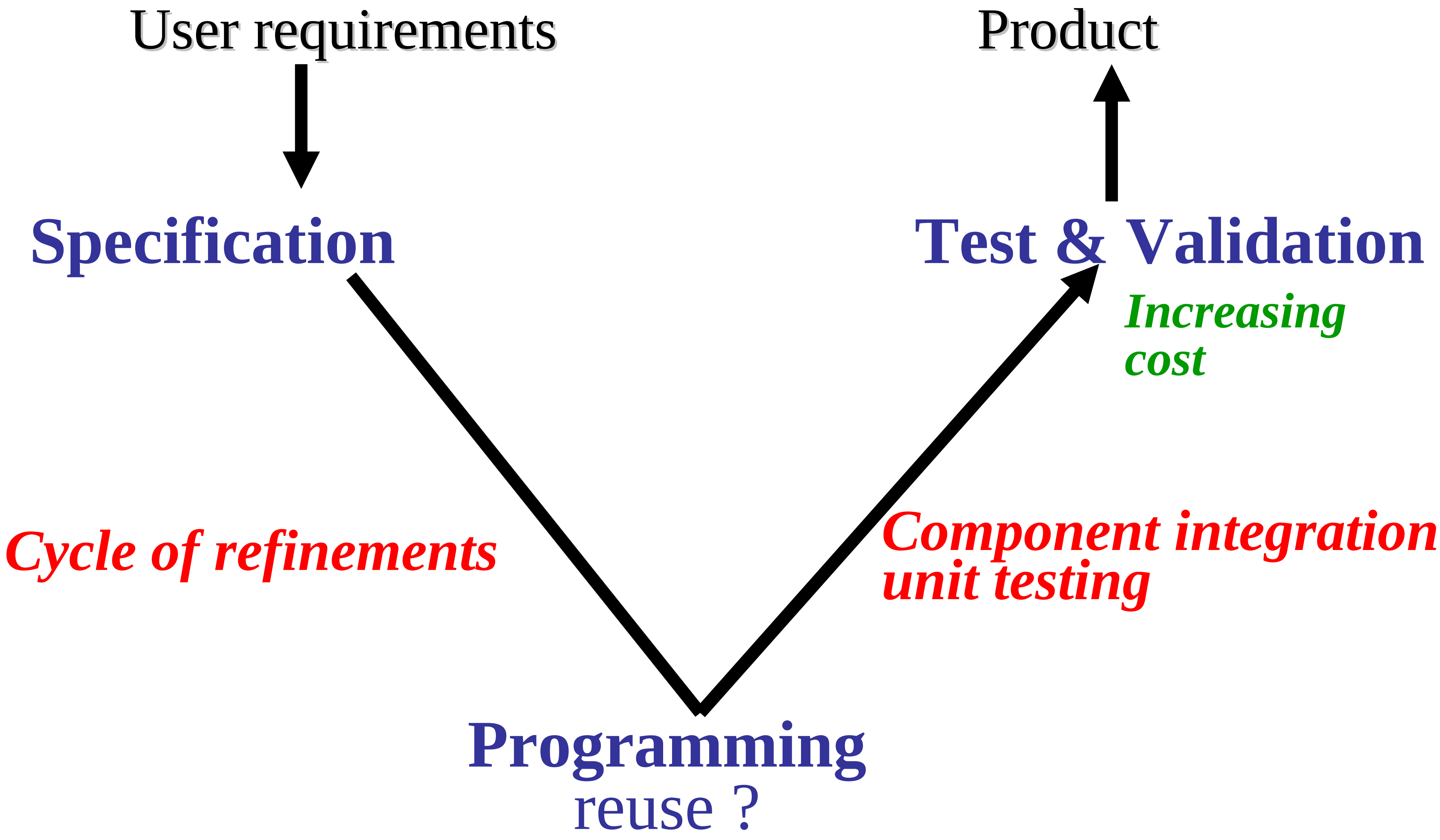
Goals of (semi) Formal Methods

- Develop programs and systems as mathematical objects
- Represent them (syntax)
- Interpret/Execute them (semantics)
- Analyze / reason about their behaviours

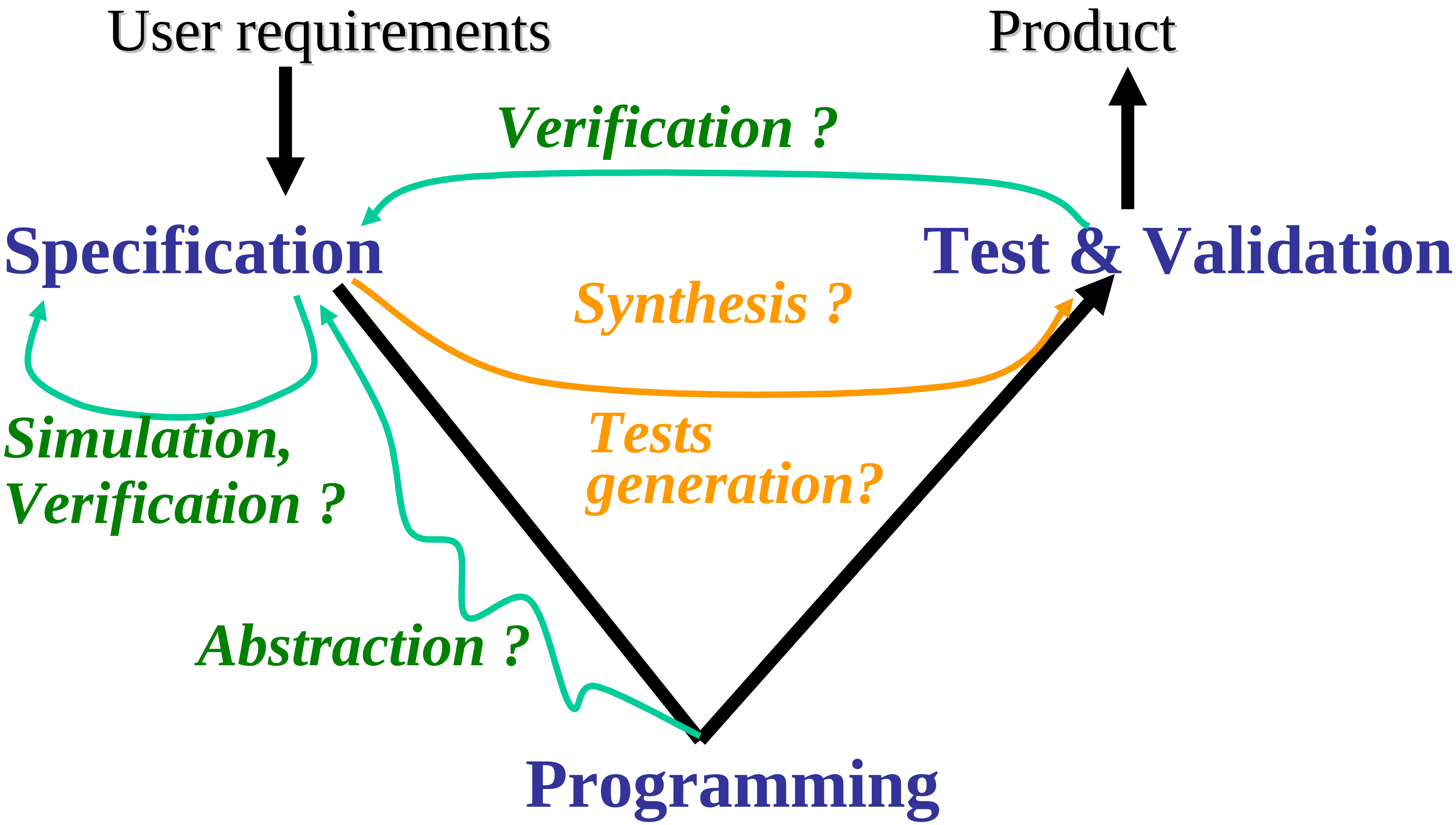
  (algorithmic, complexity, verification)
- In addition to debug, using exhaustive tests and property checking.

Software engineering (ideal view)

- Requirements informal
  - User needs, general functionalities.
  - incomplete, unsound, *open*
- Detailed specification formal ?
  - Norms, standards?..., at least a reference
  - Separation of architecture and function. *No ambiguities*
- development
  - Practical implementation of components
  - Integration, deployment
- Tests (units then global) *vs* verification ?
  - Experimental simulations, certification



V cycle (utopia)



Benefits from formal methods ?

automatisation?

Developer Needs

- Notations, syntax
  - textual
  - graphical (charts, diagrams…)
- Meaning, semantics
  - Non ambiguous signification, executability
  - interoperability, standards
- Instrumentation analysis methods
  - prototyping, light-weight simulation
  - verification

How practical is this ?

- Currently an utopia for large software projects, but :
  - Embedded systems
    - Safety is essential (no possible correction)
  - Critical systems
    - Safety, human lives (travel, nuclear)

      Ligne Meteor, Airbus, route intelligente
    - Safety, economy (e-commerce, cost of bugs)

      Panne réseau téléphonique US, Ariane 5
    - Safety, large volume (microprocessors)

      Bug Pentium

Industry succes-stories

- Model-checking for circuit development
  - Finite systems, mixing combinatory logics with register states
- Specification of telecom standards
- Proofs of Security properties for Java code and crypto-protocols.
- Certification of embedded software (trains, aircafts)

Semantics: definition, motivations

- **Give a (formal) meaning to words, objects, sentences, programs…**

Why ?

- Natural language specifications are not sufficient
- A need for understanding languages: eliminate ambiguities, get a better confidence.
- Precise, compact and complete definition.
- Facilitate learning and implementation of languages

Formal semantics, Proofs, and Tools

- **Manual proofs are error-prone !**
- **Tools for Execution and Reasoning**
  – semantic definitions are input for meta-tools
- **Integrated in the development cycle**
  – consistent and safe specifications
  – requires validation (proofs, tests, …)
- **Challenge:**

Expressive power versus executability...

Concrete syntax, Abstract syntax, and Semantics

- **Concrete syntax:**
  - scanners, parsers, BNF, ... many tools and standards.
- **Abstract syntax:**
  - operators, types, => *tree representations*
- **Semantics:**
  - based on abstract syntax
  - static semantics: typing, analysis, transformations
  - dynamic: evaluation, behaviours, ...

This is not only a concern for theoreticians: it is the very basis for compilers, programming environments, testing tools, etc...

Static semantics : examples

Checks non-syntactic constraints

- compiler front-end :
  - declaration and utilisation of variables,
  - typing, scoping, … static typing => no execution errors ???
- or back-ends :
  - optimisers
- defines legal programs :
  - Java byte-code *verifier*
  - JavaCard: legal acces to shared variables through firewall

Dynamic semantics

- Gives a meaning to the program (a semantic value)
- Describes the behaviour of a (legal) program
- Defines a language interpreter

  |- e -> e ’

  let i=3 in 2*i   -> semantic value = 6

- Describes the properties of legal programs

The different semantic families (1)

- **Denotational semantics**
  - mathematical model, high level, abstract
- **Axiomatic semantics**
  - provides the language with a theory for proving properties / assertions of programs
- **Operational semantics**
  - computation of the successive states of an abstract machine
  - used to build evaluators, simulators.

Semantic families (2)

- **Denotational semantics**
  - defines a model, an abstraction, an interpretation

    ⇒ *for the language designers*
- **Axiomatic semantics**
  - builds a logical theory

    ⇒ *for the programmers*
- **Operational semantics**
  - builds an interpreter, or a finite representation

    ⇒ *for the language implementors*

Program of the course: 1: Semantic Formalisms

- Semantics and formal methods:
  - motivations, definitions, examples
- Operational semantics, behaviour models : represent the complete behaviour of the system
  - CCS, Labelled Transition Systems
  - Equivalences

Operational Semantics (Plotkin 1981)

- Describes the computation
- States and configuration of an abstract machine:
  - Stack, memory state, registers, heap...
- Abstract machine transformation steps
- Transitions: current state -> next state

Several different operational semantics

Natural Semantics : big steps (Kahn 1986)

- **Defines the results of evaluation.**
- **Direct relation from programs to results**

env |- prog => result

  - env: binds variables to values
  - result: value given by the execution of prog

Reduction Semantics : small steps

describes **each elementary step** of the evaluation

- **rewriting relation** : reduction of program terms
- **stepwise reduction**: <prog, s> -> <prog', s '>
  - infinitely, or until reaching a normal form.

Differences: small / big steps

- Big steps:
  - abnormal execution : add an « error » result
  - non-terminating execution : problem
    - deadlock (no rule applies, evaluation failure)
    - looping program (infinite derivation)
- Small steps:
  - explicit encoding of non termination, divergence
  - confluence, transitive closure ->*

Natural semantics: examples (big steps)

- **Type checking** :

Terms: X | tt | ff | not t | n | t1 + t2 | if b then t1 else t2

Types: Bool, Int

- **Judgements :**

Typing: Γ |- P : τ

Reduction: Γ |- P $\Rightarrow$ v

Deduction rules

Values and expressions:

$$\Gamma \text{ |- } tt : Bool \qquad \Gamma \text{ |- } tt \Rightarrow true$$

$$\Gamma \text{ |- } ff : Bool \qquad \Gamma \text{ |- } ff \Rightarrow false$$

$$\frac{\Gamma \text{ |- } t1 : Int \qquad \Gamma \text{ |- } t2 : Int}{\Gamma \text{ |- } t1 + t2 : Int} \qquad \frac{\Gamma \text{ |- } t1 \Rightarrow n1 \qquad \Gamma \text{ |- } t2 \Rightarrow n2}{\Gamma \text{ |- } t1 + t2 \Rightarrow n1+n2}$$

Deduction rules

- Environment :

$\delta :: \{x\text{->}v\}$ |- $x \Rightarrow v$ **$\delta :: \{x : \tau\}$ |- $x : \tau$**

- Conditional :

$$\frac{\Gamma \mid\!\text{-}\; b \Rightarrow \mathbf{true} \qquad \Gamma \mid\!\text{-}\; e1 \Rightarrow v}{\Gamma \mid\!\text{-}\; \mathbf{if}\ b\ \mathbf{then}\ e1\ \mathbf{else}\ e2 \Rightarrow v}$$

Exercice : typing rule ?

Operational semantics: big steps for reactive systems **Behaviours**

- **Distributed, synchronous/asynchronous programs:**

  transitions represent communication events
- **Non terminating systems**
- **Application domains:**
  - telecommunication protocols
  - reactive systems
  - internet (client/server, distributed agents, grid, e-commerce)
  - mobile / pervasive computing

Synchronous and asynchronous languages

- Systems build from communicating components : parallelism, communication, concurrency
- Asynchronous Processes
  - Synchronous communications (rendez-vous)
    Process calculi: CCS, CSP, Lotos
  - Asynchronous communications (message queues)
    SDL modelisation of channels
- Synchronous Processes (instantaneous diffusion)
  Esterel, Sync/State-Charts, Lustre

Question on D. Caromel course: how do you classify ProActive ?

Program of the course: 1: Semantic Formalisms

- Semantics and formal methods:
  - motivations, definitions, examples
- Operational semantics, behaviour models : represent the complete behaviour of the system
  - CCS, Labelled Transition Systems
  - Equivalences

Labelled Transition Systems (LTS)

- Basic model for representing reactive, concurrent, parallel, communicating systems.
- Definition:

  < S, s0, L, T>

  S = set of states

  S0 $\in$ S = initial state

  L = set of labels (events, communication actions, etc)

  T $\subseteq$ S x L x S = set of transitions

  Notation: s1 $\xrightarrow{a}$ s2 = (s1, a, s2) $\in$ T

CCS

(R. Milner, "A Calculus of Communicating Systems", 1980)

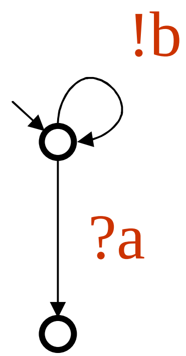
- Parallel processes communicating by Rendez-vous :

?a:!b:nil $\xrightarrow{?a}$ **!b:nil** $\xrightarrow{!b}$ **nil**

?a:P || !a:Q $\xrightarrow{\tau}$ **P || Q**

•Recursive definitions :

let rec { st0 = ?a:st1 + !b:st0 } in st0



CCS : behavioural semantics (1) Operators and rules

Inactivity — *nil (or skip)*

Action prefix — $\mathbf{a}{:}P \xrightarrow{\mathbf{a}} P$

Non deterministic choice —

$$\frac{P \xrightarrow{\mathbf{a}} P'}{P+Q \xrightarrow{\mathbf{a}} P'} \qquad \frac{Q \xrightarrow{\mathbf{a}} Q'}{P+Q \xrightarrow{\mathbf{a}} Q'}$$

CCS : behavioural semantics (2)

More operators, more rules

Emissions & réceptions are dual actions

$$\frac{P \xrightarrow{a} P'}{P||Q \xrightarrow{a} P'||Q} \qquad \frac{Q \xrightarrow{a} Q'}{P||Q \xrightarrow{a} P||Q'}$$

τ invisible action (internal communication)

$$\frac{P \xrightarrow{!a} P' \qquad Q \xrightarrow{?a} Q'}{P||Q \xrightarrow{\tau} P'||Q'}$$

Recursion :

$$\frac{[\mu X.P/X]P \xrightarrow{a} P'}{\mu X.P \xrightarrow{a} P'}$$

Local action :

Tool for forcing synchronisation

$$\frac{P \xrightarrow{a} P' \qquad a \notin \{?b, !b\}}{local\ \mathbf{b}\ in\ P \xrightarrow{a} local\ \mathbf{b}\ in\ P'}$$

Derivations (construction of each transition step)

$$\dfrac{\dfrac{\dfrac{}{?a{:}P \xrightarrow{?a} P}\ \text{Prefix}}{?a{:}P \,||\, Q \xrightarrow{?a} P \,||\, Q}\ \text{Par-L} \qquad \dfrac{}{!a{:}R \xrightarrow{!a} R}\ \text{Prefix}}{(?a{:}P \,||\, Q) \,||\, a!{:}R \xrightarrow{\tau} (P \,||\, Q) \,||\, R}\ \text{Par-2}$$

Par-2(Par_L(Prefix), Prefix)

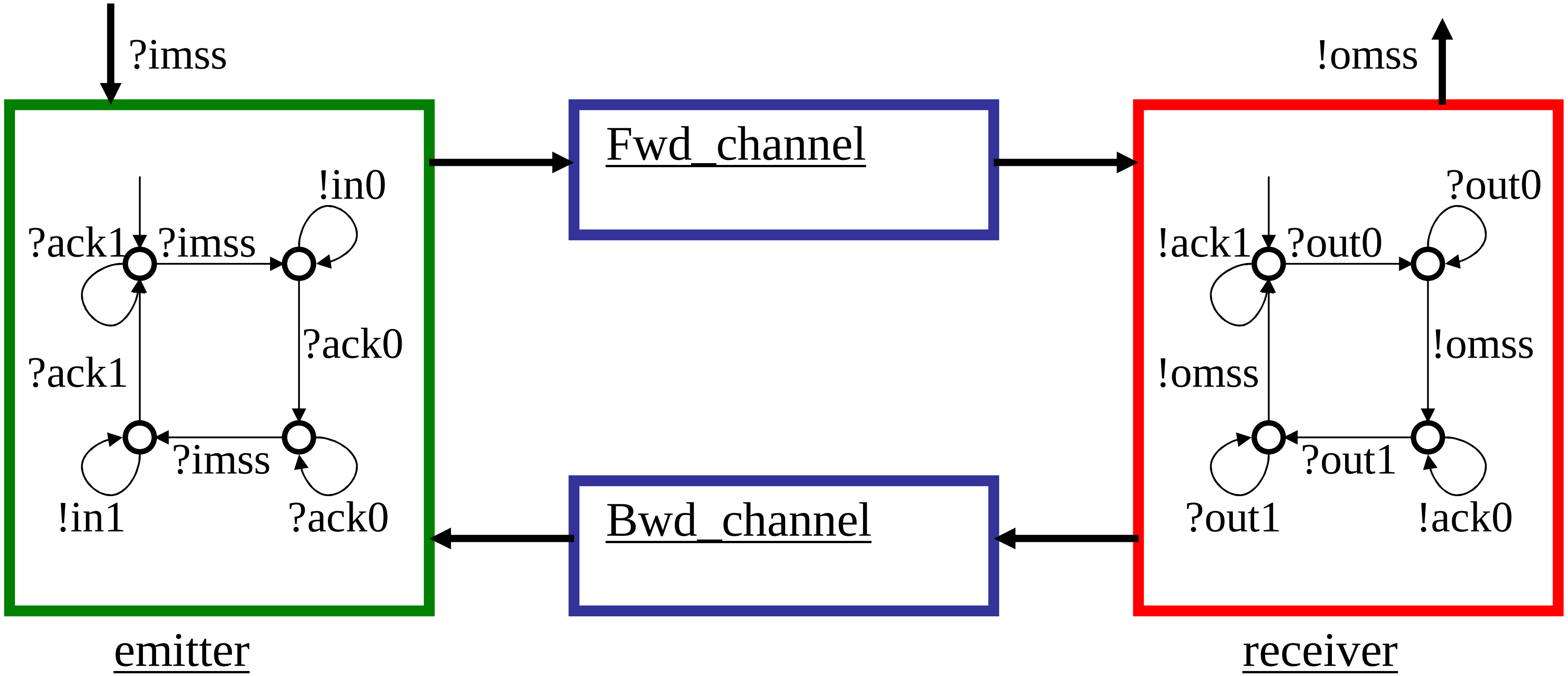
One amongst 3 possible derivations

Another one :

Par-L(Par_L(Prefix))

$$(?a{:}P \,||\, Q) \,||\, !a{:}R \xrightarrow{?a} (P \,||\, Q) \,||\, !a{:}R$$

Example: Alternated Bit Protocol



Write in CCS ?

Hypotheses: channels can loose messages

Requirement:

the protocol ensures no loss of messages

Example: Alternated Bit Protocol (2)

- **emitter** =

```
let rec {em0 = ?ack1 :em0 + ?imss:em1
  and em1 = !in0 :em1 + ?ack0 :em2
  and em2 = ?ack0 :em2 + ?imss :em3
  and em3 = !in1 :em3 + ?ack1 :em0
  }
  in em0
```

- **ABP =** local {in0, in1, out0, out1, ack0, ack1, …}
  in emitter || Fwd_channel || Bwd_channel || receiver

Example: Alternated Bit Protocol (3)

Channels that loose and duplicate messages (in0 and in1) but preserve their order ?

- Exercise :
  1) Draw an LTS describing the loosy channel behaviour
  2) Write the same description in CCS

Program of the course: 1: Semantic Formalisms

- Semantics and formal methods:
  - motivations, definitions, examples
- Operational semantics, behaviour models : represent the complete behaviour of the system
  - CCS, Labelled Transition Systems
  - Equivalences

Behavioural Equivalences

- Intuition:
  - Same possible sequences of observable actions
  - Finite / infinite sequences
  - Various refinements of the concept of observation
- Definition: Trace Equivalence

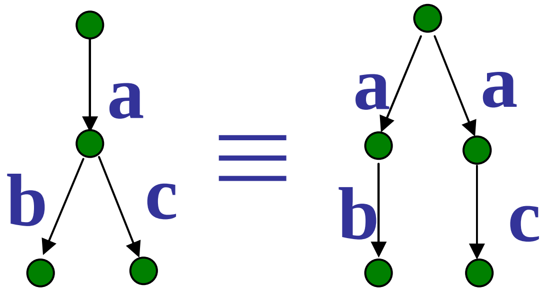
For a LTS (S, s0, L, T) its Trace language ***T*** is the set of finite sequences $\{(t = t_1, \ldots, t_n$ such that $\exists s_0,\ldots,s_n \in S^{n+1,}$ and $(s_{n-1},t_n,s_n) \in T\}$

Two LTSs are Trace equivalent iff their Trace languages are equal.

Corresponding Ordering: Trace inclusion

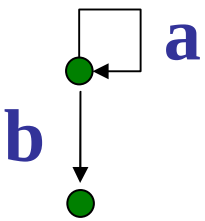
Trace Languages, Examples

1. Those 2 systems are trace equivalent:



$\boldsymbol{T}$ = {(), (a), (a,b), (a,c)}

2. A trace language can be an infinite set:



$\boldsymbol{T}$ = {(), (a), (a,a), (a,…,a),…
(a,b), (a,a,b), (a,a,…,a,b), …}

Bisimulation

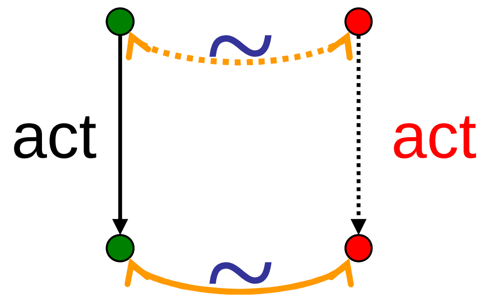
- **Behavioural Equivalence**
  - non distinguishable states by observation:

    two states are equivalent if for all possible transitions labelled by the same action, there exist equivalent resulting states.

- **Bisimulations**

  R ⊆ SxS is a bisimulation iff
  - It is a equivalence relation
  - ∀(p,q) ∈ R,

    (p,l,p') ∈ T => ∃ q'/ (q,l,q') ∈ T and (p',q') ∈ R



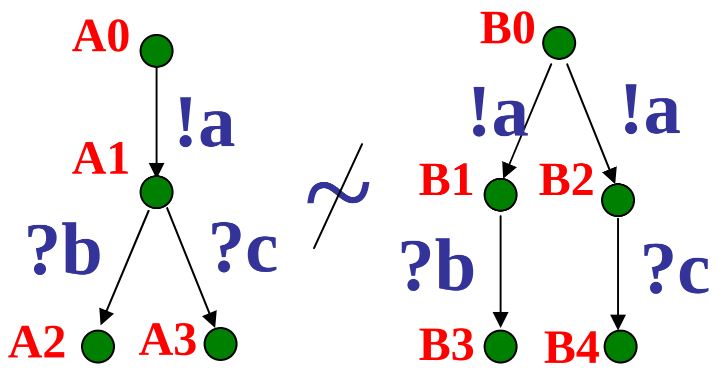
- **~ is the coarsest bisimulation**

  2 LTS are bisimilar iff their initial states are in ~

  quotients = canonical normal forms

Bisimulation (3)

- More precise than trace equivalence :



No state in B is equivalent to A1

- Preserves deadlock properties.

Bisimulation (4)

- Congruence laws:

  P1~P2 => a:P1 ~ a:P2 (∀ P1,P2,a)

  P1~P2, Q1~Q2 => P1+Q1 ~ P2+Q2

  P1~P2, Q1~Q2 => P1||Q1 ~ P2||Q2

  Etc…

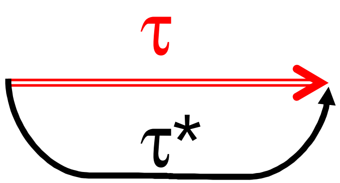
- ~ is a congruence for all CCS operators :

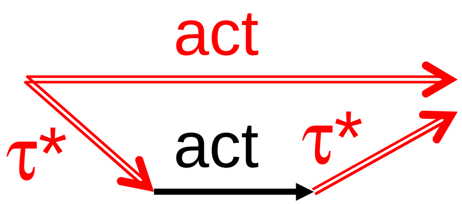
  for any CCS context C[.], C[P] ~ C[Q] <=> P~Q

  Basis for compositional proof methods

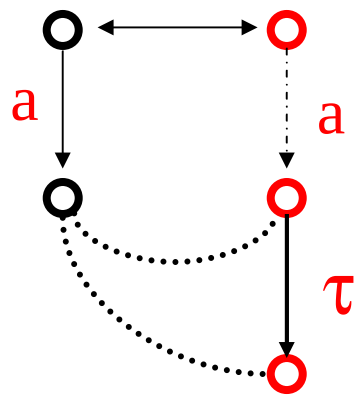
Observational Equivalences

- Weak bisimulation
  - Abstraction: hidden actions
  - allows for arbitrary many internal actions



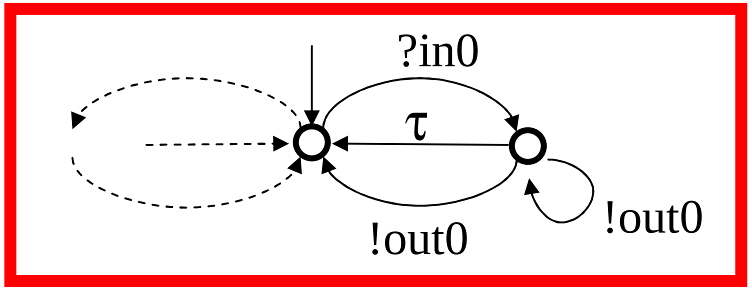


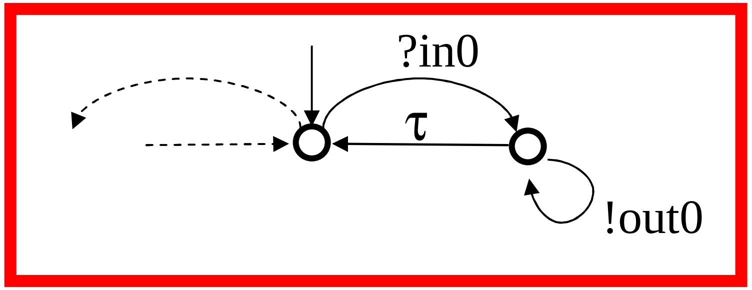
- Branching bisimulation
  - … only staying in equivalent states



Still existence of a canonical minimal automata
Computation is polynomial

Exercice 2 : Bisimulations





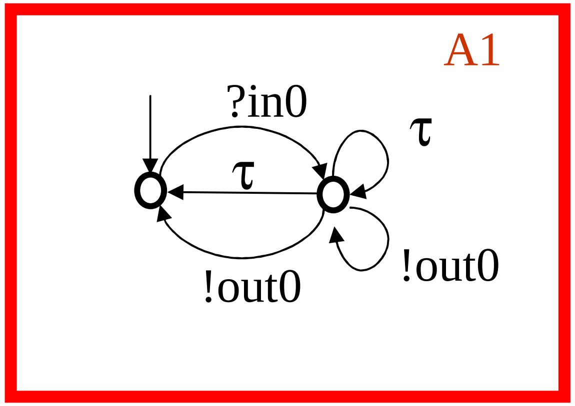
Are those LTSs equivalent by:

- Strong bisimulation?

- Weak bisimulation ?

In each case, give a proof.

Exercice 3 : Bisimulation

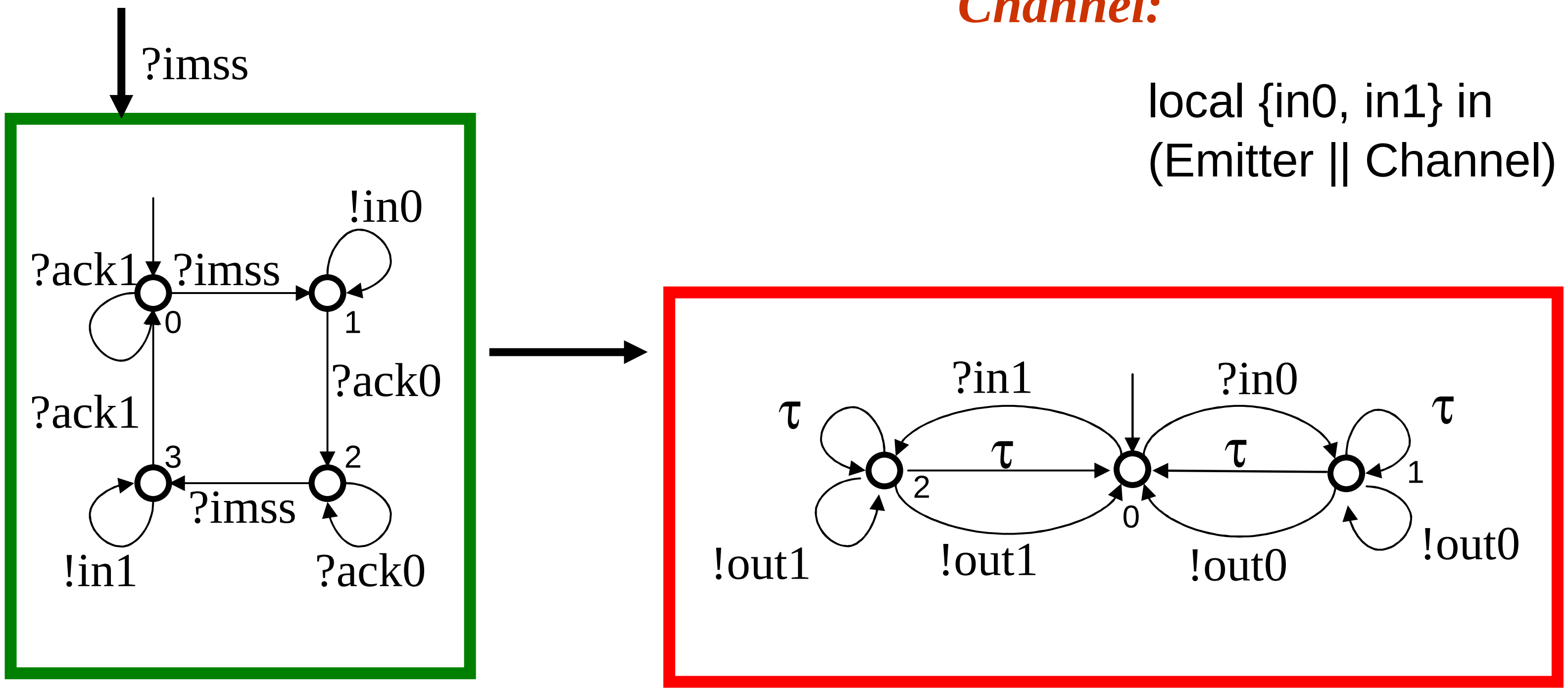


- Exercice :
  1) Compute the strong minimal automaton for A1.
  2) Compute the weak minimal automaton for A1.

Exercice 4 : Synchronized Product

Compute the synchronized product of the LTS representing the ABP emitter with the (forward) Channel:

local {in0, in1} in
(Emitter || Channel)



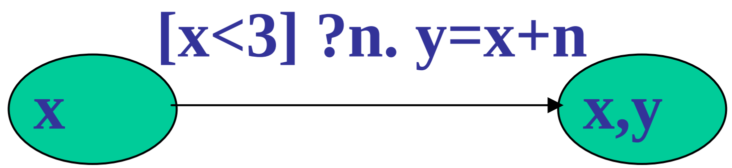
Automatas with data

from state<i>

provided guard_cond(vars)

then execute body

goto state<j>



- We need add: if_then_else : tree of successor states

  guards and conditions on external signals

  local variables (scoping)

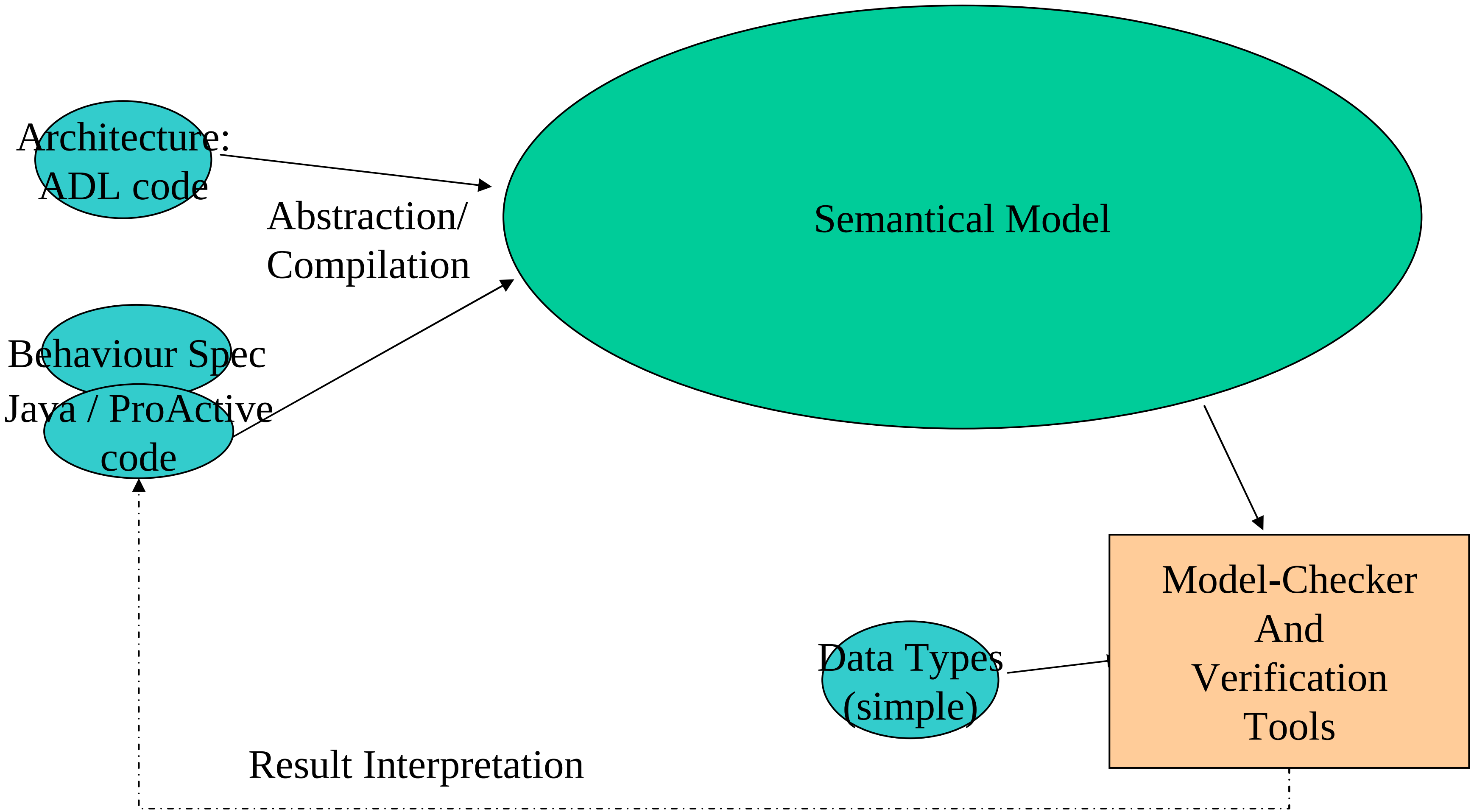
➡ ***Graphical specifications languages : SDL, Statecharts, etc.***

The Dream

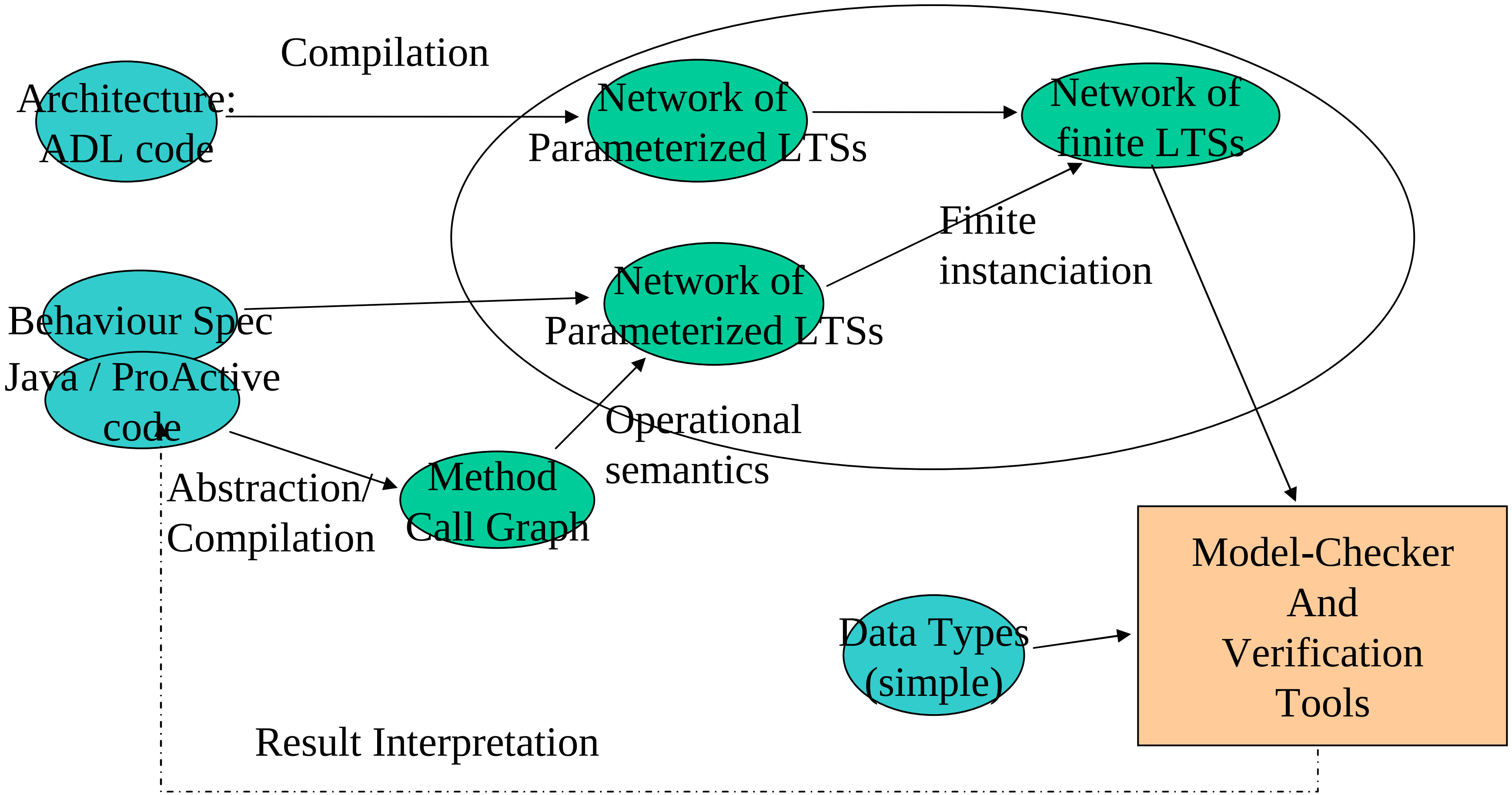
Provide Analysis and Verification Tools to the (non-specialist) programmer

- Specification Language (textual or graphical)
- Code analysis tools
- Automatic Model-Checking

Tool Set (future…)



Tool Set (future…)



Next courses

3) Software Components

- Fractal : main concepts
- Deployment, management, transformations
- Specification of components

2) Application to distributed applications

- ProActive : active object and distributed components
- Behaviour models
- Tools : build an analysis and verification platform

www-sop.inria.fr/oasis/Eric.Madelaine

⇨ **Teaching/RSD-2006**