Software Engineering for Real-Time Systems: the Synchronous Approach can Help

Florence Maraninchi
Pr. INPG/ENSIMAG and Vérimag laboratory
Grenoble, France
www-verimag.imag.fr

VERIMAG

Study and develop specification formalisms and validation techniques for critical software and systems

— A good balance between fundamental, experimental and applied research

— Long term cooperations with academic and industrial partners

INPG/ENSIMAG

INPG : Institut National Polytechnique de Grenoble
www.inpg.fr

9 schools, including ENSIMAG
Applied Math. and Computer Science
140 students / year
50 professors and assistant professors.

Personal Data — Teaching

- Initiation to Computer Science
- Algorithmics and Programming Languages (functional, OO, imperative, ...)
- Data Bases
- Elements of circuit design, basics of operating systems
- Object-Oriented Analysis and Design (UML)
- Software Analysis and Formal Verification
- Synchronous Languages

Personal Data — Research

Analysis, design, implementation and validation of critical software, from the *language* point of view

- Argos: an automaton-based synchronous language with the syntax of Statecharts
- Multi-language programming of reactive systems
- Quantitative time and non-determinism in synchronous languages
- Mode-automata
- ...

This talk...

- The kind of computer systems we are interested in
- The synchronous approach in general, and Lustre
- The SeeS project at Verimag

Software Engineering for Embedded Systems

The kind of computer systems

we are interested in

Real-Time, Embedded, Safety-Critical Software and Systems

Characteristics that require specific techniques and tools:

— Criticity
— Reactivity and the notion of environment, time
— Centralized or distributed systems
— Links with control theory
— The frontier between data and control
— All-purpose languages are not appropriate

Criticity

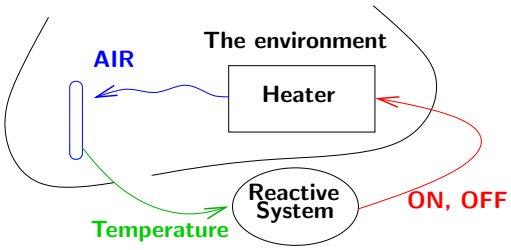
Contexts: avionics, nuclear plants, trains, lifts, ...

1) Errors are expensive
(human lives, damage to the environment, money...)

AND
2) When the error has occurred, it's too late!

$\Longrightarrow$ this requires *static analysis* techniques and tools

Reactivity, environment, time (1)



Reactivity, environment, time (2)

The environment is a strong constraint
(it cannot wait, nor re-emit events)
BUT it can be modeled.

— If the heater has been switched on, the temperature increases
— Trains do not land on tracks
— ...

Reactivity, environment, time (3)

We should be able to describe the environment, and to take it into account when validating reactive systems

The description should be:
— Accurate enough
— As simple as possible

The power of physics <----------> models that can be tackled by computers

Centralized or Distributed Systems (1)

Synchronous: all parts of the system share a global clock

Asynchronous: no common notion of time

Quasi-Synchronous (P. Caspi): components have their own clocks, but *they don't differ too much*

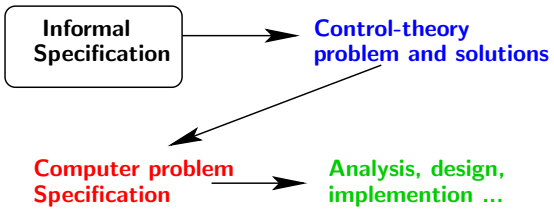
— Synchrony is far simpler than asynchrony
— Perfect Synchrony forbids distributed Systems
⟶ Choose the appropriate model

Centralized or Distributed Systems (2)

Examples:
— The computers of an aircraft: quasi-synchronous
— The emitter and the receiver in a protocol: asynchronous
— A digital watch : synchronous
— The emitter of a protocol, alone: synchronous !

⟶ distinguish between logical and physical parallelisms

Links with Control-Theory



Data and Control (1)

No real "data" problem (large volume, storage or communication problems, querying, etc.)

But: a regulation system (for instance) makes decisions on numerical data crossing thresholds.

Data and Control (2)

Some facts
Imperative style languages :
data and control are clearly separated

Some validation techniques (model-checkers) rely on a clear notion of *control state* (sometimes enumerated)

High level language	model used by validation tools
Control structure	states (case analysis)
Data	"variables" (often abstracted)

Data and Control (3)

But:
— not all languages have a clear control/data distinction
— Case analysis should not be given by the syntax

Data and Control should be treated the same as long as possible: everything described symbolically, expand when needed

Need for powerful validation techniques that can take numerical aspects into account... necessarily approximate techniques, since problems are usually undecidable.

Language needs

— MORE than general-purpose languages :
i/o, parallel and time constructs

— LESS than general-purpose languages :
no dynamic errors (even properly trapped by exceptions)

Languages adapted to engineers practise
(e.g., control engineers don't speak ML).

We need...

- Domain-specific languages
  - ADD description of i/o, time and parallelism
  - REMOVE sources of dynamic errors
- ... with formal semantics
- Development environments offering:
  - compilers into efficient code (e.g., C)
  - Automatic validation tools
    (assisted debug, test, proof, ...)
- Dedicated development methods

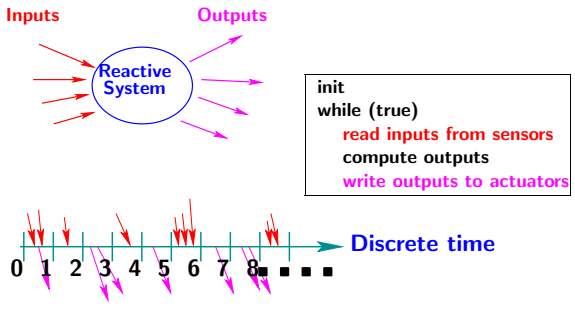
The Synchronous Approach

1) Generalities

History (in France)

- Control-theory and computer science people met (Lustre in Grenoble, Esterel in Sophia Antipolis, Signal in Rennes, ...)
- Definition of domain-specific languages
  - dataflow style (Lustre, Signal)
  - imperative style (Esterel)
- Corresponds to what people have been doing in industry for a long time (Saga at Schneider Electric, SAO at Aérospatiale, ...)

The Synchronous view of the World



Quality of the code: the "real-time" correctness condition

Maximum time between two successive samplings of the inputs =
WCET of one execution of the loop body

maximum reaction time $<$ minimum environment delay

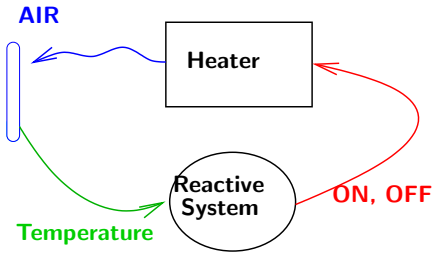
Synchronous Language: definition

(1) A High-level, modular, domain-specific language :
i/o and parallelism are easy to describe
don't write the loop yourself

(2) Can be compiled into efficient cyclic-executives
the "parallel" construct is compiled

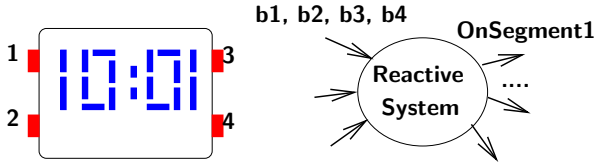
(3) Has a formal (mathematical) semantics

Ex: A Regulation System



actuator$_n$ = F (sensor$_n$, sensor$_{n-1}$, ..., sensor$_0$)

Ex: An Event-Driven System



OnSegment1$_n$ = F (buttons$_n$, buttons$_{n-1}$, ... buttons$_0$)

Typical Program (embedded code)

```
Initializations
while true
      Get inputs i
      Compute outputs o        ← Reactive kernel
      Emit outputs o
```

⇒ Execution cycles =
instants of a discrete logical time N
At instant $n \in N$, input i_n and output o_n

Input/output Relations

- General case : $\forall n,\ o_n = f(i_0, i_1, ..., i_n)$
  Remarks : { o_n may not depend on the *future* $i_{n+...}$; o_n may depend on i_n ; The history of inputs is not bounded }
- Combinational circuits : $\forall n,\ o_n = f(i_n)$
  No memory
- Bounded-Memory Systems (Sequential Circuits) :
  $\forall n,\ o_n = f(Abs(i_0, i_1, ..., i_n))$
  Where Abs is an *abstraction* function,
  such that $\exists B \mid Image(Abs) \mid < B$

Bounded-Memory Systems

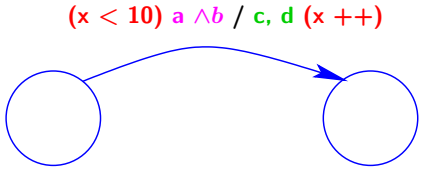
Initialization of the *memory* M
while true

Invariant :	$M = Abs(i_0, i_1, ..., i_{n-1})$
Get input	i_n
Compute output	$o_n = f(M, i_n)$
Update memory	$M = g(M, i_n)$
Property here :	$M = Abs(i_0, i_1, ..., i_n)$
Emit output	o_n

Languages for the Reactive kernel

- dataflow (lustre, Signal) : Systems of Equations between the *flows* of values of i, o, M
- Control Structures (Esterel) : parallel composition, watchdogs, ...
- Automata (Statecharts, Argos) : explicit memory M and f (output function) and g (transition function) given in extension
- Automata+dataflow (Mode-Automata) : dataflow equations attached to the states of an automaton.

Common semantics = interpreted automata



— A deterministic finite-state automaton (locations)
— A finite set of variables (any type) like x
— Finite sets of input and output signals.

memory = location + values of the variables

The synchronous paradigm is not really new

Classical in synchronous circuits

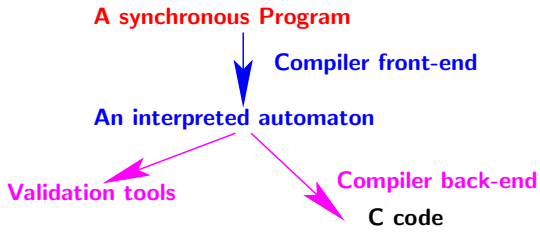
- synchronous communicating Mealy machines
- dynamic Boolean equations
- gate and latch networks

Classical in control engineering

data-flow synchronous formalisms

- differential or finite difference equations
- block-diagrams, analog networks

But... validation, code generation



Automata may be explicit
(enumerated states and transitions)
or implicit (equations - cf. Lustre)

The Synchronous Approach

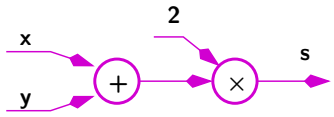
2) Lustre

Basic ideas

- programmers of reactive systems are often control engineers
- their classical description tools are declarative (equations, data-flow networks)

Language style

The diagram



and the equation `s = 2 * (x+y)`

both mean $\forall t, s_t = 2(x_t + y_t)$
or at each cycle $n, s_n = 2(x_n + y_n)$

Variables denote flows of values

Each variable (or constant) x denotes a flow:
the infinite sequence of values $x_0, x_1, \ldots, x_n, \ldots$.

x_n is the value of x at the n-th execution cycle of the program.

Program are sets of equations

Exactly one equation for each output or local variable.

x = exp means $\forall n, x_n = \text{exp}_n$

Example:
s = 2*(x+y) or s = 2*sum; sum = x+y

Usual arithmetic, boolean, conditional operators are available (implicitly assumed to operate pointwise on flows)

+ Temporal operators

"Pre" refers to the past: $\text{pre(x)}_n = \text{x}_{n-1}$

Initialization operator: $(\text{x -> y})_n = \begin{cases} \text{x}_0 & \text{if } n = 0 \\ \text{y}_n & \text{otherwise} \end{cases}$

Example: `x = y -> (pre(x) + y)`

means $x_0 = y_0 \qquad x_n = x_{n-1} + y_n$

or $x_n = \sum_{i=0}^{n} y_i$

Example

C counts the occurrences of E,
and is reset when R:

```
C = 0 -> if R then 0
         else if E then pre(C) + 1
         else pre(C);
```


+ nodes (user defined operators)

Node = a function of flows
or: encapsulation of a subnetwork into a new operator

```
node Counter
(R, E: bool)
returns (C: int) ;
let
  C = 0 ->
      if R then 0
      else if E then
        pre(C) + 1
      else pre(C);
tel
```

```
node TimeKeeper (S: bool)
returns (NbS, NbM, NbH: int);
var M, H: bool;
let
  NbS = Counter(M, S);
  M = S and pre(NbS) = 59;
  NbM = Counter(H, M);
  H = M and pre(NbM) = 59;
  NbH = Counter (pre(NbH)=23,H);
tel
```


From Lustre to sequential code (1)

Single loop (implicit automaton)

- sort the variables according to their instantaneous dependencies
- choose a suitable set of memories
  For instance, `y = pre(pre(x))` is the same as:
  `aux = pre(x) ; y = pre(aux)`

Example

```
x    = 0 ->
       if edge then
         (pre(x) + y)
       else pre(x) ;
edge = c ->
       (c and not pre(c)) ;
```

```
_init := true ;
foreach step do
   read(c, y) ;
   if _init then
      { edge := c ; x := 0}
   else {
      edge := c and not _pc ;
      if edge then x := x+y ;
   }
   _init := false ; _pc := c ;
end for
```

Compilation into explicit automata

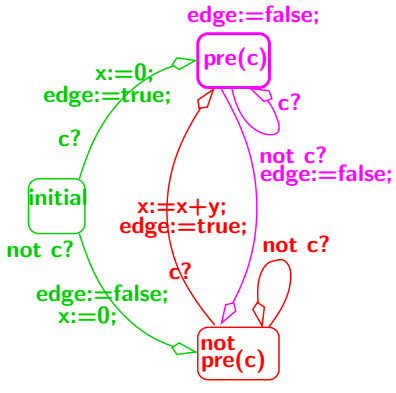
- First way of compiling Esterel
- Also applied to data-flow languages
- Basis for verification methods
  (via the notion of control automaton)

For Lustre : choose the variables whose values are expanded into explicit states.
Usual choice: the Boolean memories.

Very efficient code (time)
But a potential exponential growth of the number of states.

From Lustre to explicit automata: example

```
x    = 0 ->
       if edge then
         (pre(x) + y)
       else pre(x) ;
edge = c ->
       (c and
         not pre(c)) ;
```



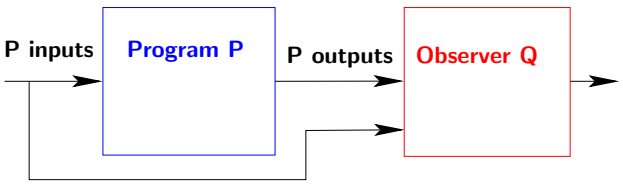
The Synchronous Approach

3) Program Validation:

Test, model-cheking, ...

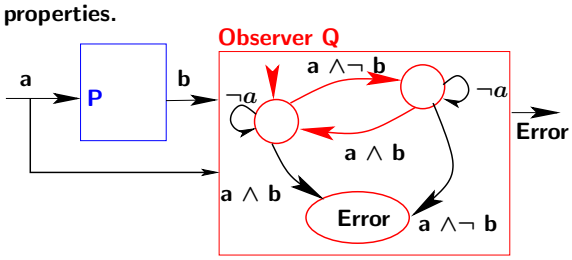
The "Observer" principle (1)

Observer of P: another synchronous program Q connected to P *without feedback*



The "Observer" principle (2)

An observer transforms path (safety) properties into state properties.



Q means "P emits b every two a's"

The "Observer" principle (3)

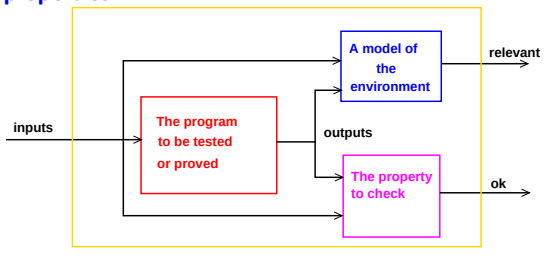
In Lustre: ordinary dataflow connection
In Esterel: parallel composition of two components communicating via synchronous broadcast.

The principle is valid in all synchronous languages (due to the synchronous broadcast).
Impossible with rendez-vous.

— Easy expression of safety properties
— Observers are executable

The general validation scheme

2 observers: Hypothesis on the environment and safety properties

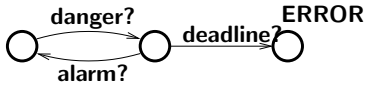


Examples

Environment Hypothesis:

T increases: `true -> (T > pre (T))`

Safety property: each occurrence of danger involves an alarm before the next deadline



Can use any synchronous language (but Lustre is a temporal logic of the past).

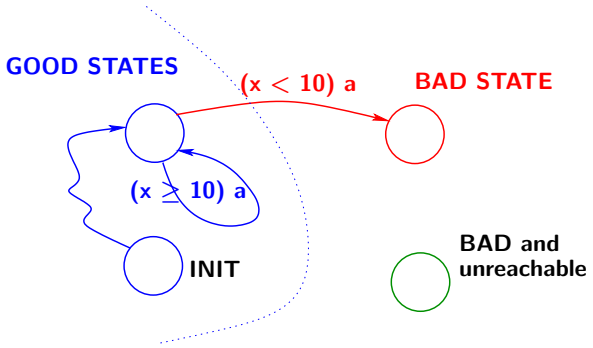
The verification problem

In the product $P \mid\mid E \mid\mid \Phi$, the error state is unreachable with a path that emits "relevant" on each transition.

Verification based on the control automaton is:

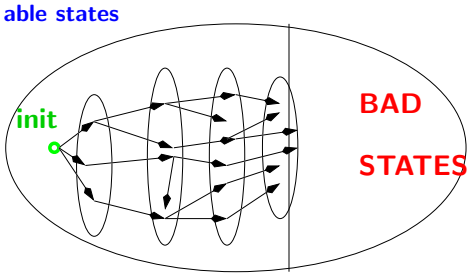
- exact for pure control programs
- conservative in general (safety properties are preserved by abstractions)

Abstraction: an example



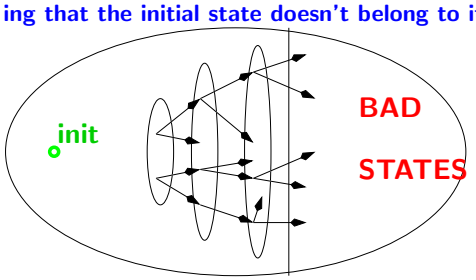
Forward symbolic search

Symbolic construction of the set of ("relevantly") reachable states



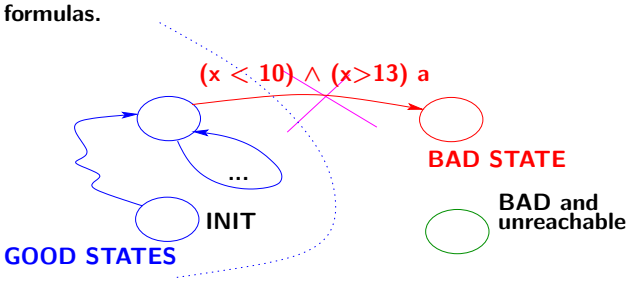
Backward symbolic search

Symbolic construction of the set of states that can lead to violation, without emitting irrelevant on the way, checking that the initial state doesn't belong to it.



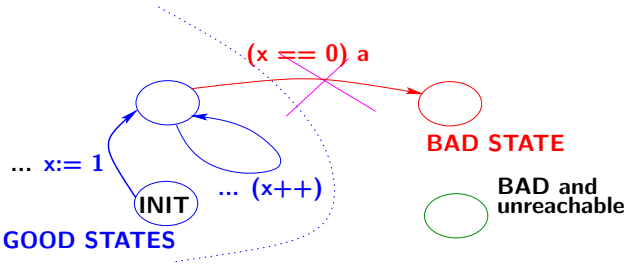
Handling numerical properties in LESAR

A bit of knowledge about the satisfiablity of arithmetical formulas.



Lesar - Pascal RAYMOND, Verimag

Handling Dynamical Aspects: the tool NBAC



Nbac - Bertrand Jeannet, Verimag

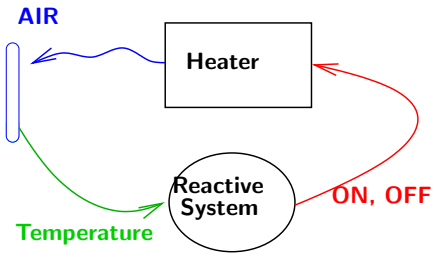
Automatic test-case generation: LURETTE

Given:

- A synchronous program to be tested (black box)
- A Lustre observer modeling the environment
- A Lustre observer for the oracle

Lurette generates and runs arbitrarily long test sequences, satisfying the environment observer, while checking the oracle observer.

Example: the heater



Example: the heater

The controller C should maintain the temperature u between 17° and 20° by turning a heater on and off.

When the heater is running, the temperature u increases with $0.2 \leq du/dt \leq 0.5$; otherwise it decreases with $-0.3 \leq du/dt \leq -0.1$.

Initially, the heater is turned off, and $18° \leq u \leq 19°$

Lustre observer for the environment

```
node Hypothesis (on, off: bool; u: real)
returns (relevant: bool);
var dudt: real; heating: bool;

  relevant = (u>=18 and u<=19) ->
             if heating then (dudt>=0.2 and dudt<=0.5)
             else (dudt>=–0.3 and dudt<=–0.1);

  dudt =     (u – pre(u));

  heating =  false -> if pre(on) then true
                      else if pre(off) then false
                      else pre(heating);
```

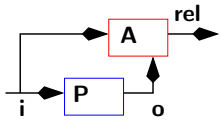

Lustre observer for the oracle

```
node Property (on, off: bool; u: real)
returns (ok: bool);
let
   ok = (u>=17 and u<=20);
tel
```


Relevant test sequences

Definition:
relevant test sequences =

$(i_1, i_2, \ldots, i_n)$



involving outputs

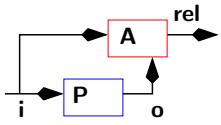
$(\langle o_1, \mathsf{rel}_1 \rangle, \langle o_2, \mathsf{rel}_2 \rangle, \ldots, \langle o_n, \mathsf{rel}_n \rangle)$

with $\mathsf{rel}_i =$ **true**, $\forall i \in [1, n]$.

How are the input sequences generated?

First attempt: Observer as an acceptor

– randomly choose input i
– submit it to A
– in case of refusal, choose again.



Main problem

the probability to randomly choose a relevant input can be very small (e.g., for X = Y)

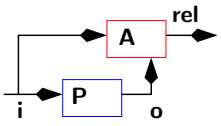
How are the input sequences generated?

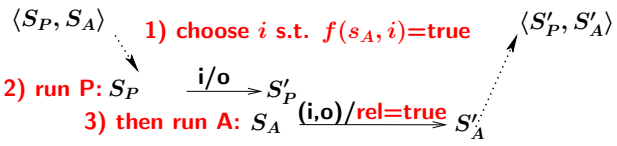
Observer as a generator

Global state: $\langle S_P, S_A \rangle$

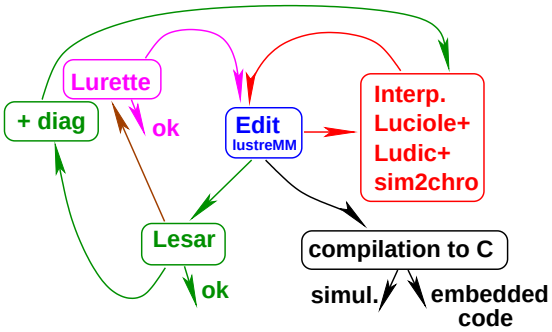
rel $= f(S_A, i)$

$S'_A = g(S_A, i, o)$





The Lustre factory



Industrial Uses of Lustre

Aerospatiale (Airbus)
Honeywell
Eurocopter
Hong-Kong subway
Schneider-Electric (nuclear plants)
EDF (Electricité de France)
...

The SeeS Project at Verimag

Software Engineering for Embedded Systems

"A posteriori" verification is an utopy

"A posteriori" = at the end of the development.

- Size of the complete code
- Complexity of the properties
- How to keep track of the design decisions?
- ...

But: a lot of efficient validation tools have been developed.

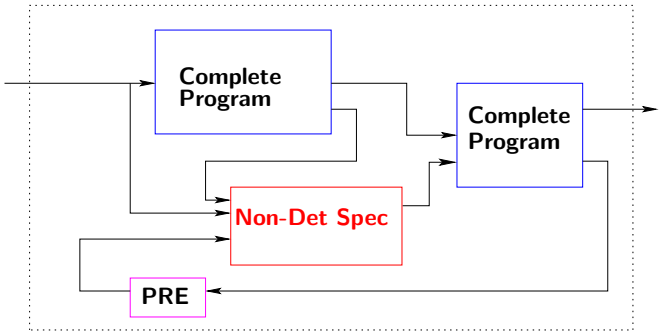
Automatic validation as early as possible

$\longrightarrow$ Try to use all the validation tools that deal with various representations of programs, in the early phases of development.

SeeS objectives

- Guided Debugging
- Early execution of non-deterministic specifications
- Design-by-contract
- Object-oriented features in synchronous languages

Early execution of non-deterministic specifications



Non-deterministic specifications

(deterministic) Programs: o **is function of** i **and** M

Non-deterministic Specifications:
there is a relation R **between** o, i **and** M.

Lustre is deterministic, but a non-det specification may be described by an observer (acceptor instead of generator).

Synchronous programming and the "design-by-contract" principle

- **Easy expression of contracts, including time, in the same language**
- **Mechanisation of proof obligations due to contracts**

Contracts

Example (Java syntax) :

```
class C {
      int m ;
      public Tr1 m1 (Ti1 i) {
         // require:  0 < i < m
         // guarantee:  m' = m+1
         ...  code for m1 ...
      }
}
```

JContract generates defensive code,
Eiffel has language-support for contracts...

Timed Contracts?

For each instance of class `C`, method `m1` should be called before method `m2` can be called.

```
class C {
      bool m1HasBeenCalled = false ;
      public T1r m1 (Ti1 i) {
         // require:  ...
         // guarantee:  m1HasBeenCalled = true
      }
      public Tr2 m2 (Ti2 i) {
         // require:  m1HasBeenCalled = true
         // guarantee:  ...
      }
}
```


Contracts for real-time components

We need timed contracts:

-> Use Lustre as a language for contracts

We need to validate systems early (can't rely on defensive code raising exceptions at runtime):

-> Use Lurette to perform early execution of non-deterministic specifications given by their contracts only.