Soft. Eng & Compilation

The synchronous approach

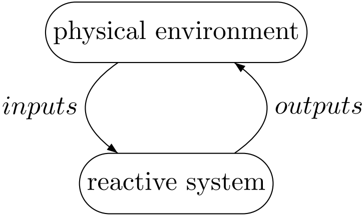
Laure Gonnord

University of Lyon/ LIP

CR10 - M2ENSL 2017

Plan

Reactive system



(CC-BY-SA Captainm/Wikipedia)

- React to inputs:
  - Acquire inputs on sensors;
  - Compute;
  - Produce values on actuators.
- Actions impact the environment, thus subsequent inputs;
- Response time must be *bounded*.

Programming

Functional correction

Compute the correct output values.

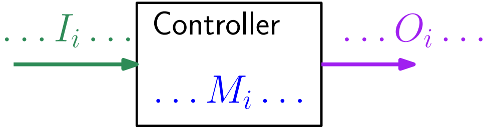
Temporal correction

Compute faster than the reactivity constraint.

Programming: the Functional Part

Remarks

- O_i depends *only* on the $I_1, I_2, \ldots, I_i$;
- Computations performed with *bounded memory* M_i.



Programming is ...

- ... identifying inputs and outputs;
- ... defining:
  - The **output function** $O_i = f(I_i, M_i)$;
  - The **transition function** $M_{i+1} = g(I_i, M_i)$

The Asynchronous approach

Parallel processes $\Rightarrow$ Concurrent multi-task implementation.

Difficulties:

- Scheduling: handle hard to predict execution times, jitter, etc;
- Inter-task communications: handle communication order (priorities, rendez-vous, semaphores, etc).

Globally non-deterministic.

The synchronous approach

Real-time is replaced by a simplified, abstract, *logical time*.

- *Instant*: one reaction of the system;
- Logical time: sequence of *instants*;
- The program describes what happens during each instant;
- *Synchronous hypothesis*: computations complete before the next instant. If so:
  - $\Rightarrow$ We can ignore time inside an instant, only the order matters;
  - $\Rightarrow$ We are only interested in how instants are chained together.

Synchronous languages vs others

Advantages:

- Semantics defined formally $\Rightarrow$ enables formal proofs and provable compilation;
- High abstraction level $\Rightarrow$ less work for the programmer, more for the compiler;
- Bounded memory and execution time;
- Barely needs an OS.

Disadvantages:

- Produced code less efficient than hand-written code;
- Synchronous hypothesis hard to ensure (WCET, distributed systems);
- Not well-suited for multi-rate systems.

Plan

Flows and clocks

- LUSTRE is a data-flow language: computations are triggered by incoming data;
- In LUSTRE, every expression and variable is a flow;
- *Flow*: infinite sequence of values + clock;
- *Clock*: defines when a flow is present (has a value).

Example

x	3	4	5	2	6	...
y	True			False	True	...

Point-wise extension of classic operators

Example

c	True	False	True	False	...
x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
x+y	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...
`if c then x else y`	x_1	y_2	x_3	y_4	...

Delay operator

- The *pre* operator denotes the previous value of a flow;
- Undefined for the first instant;
- Usually combined with the initialisation operator `->`: `x->y` is x on the first instant, y otherwise.

Example

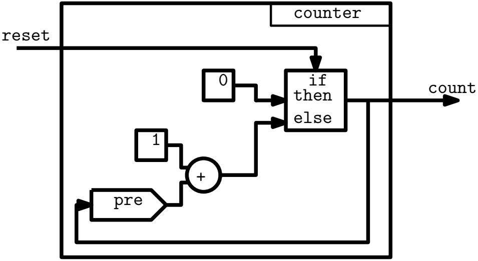
`x`	x_1	x_2	x_3	x_4	...
`y`	y_1	y_2	y_3	y_4	...
`pre x`		x_1	x_2	x_3	...
`y->pre x`	y_1	x_1	x_2	x_3	...

Structuration: nodes

- A LUSTRE program consists of a set of *nodes*;
- The *main node* is specified by the compilation line;
- Each node contains a set of *equations*, which defines output flows from input flows;
- Equations are unordered;
- Nodes can be instantiated in flow expressions (like functions).

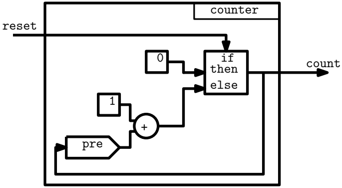
Example: A Resettable Counter

```
node counter(reset:bool) returns (count:int)
let
  count = 0 -> if reset then 0 else pre(count)+1;
tel
```



Example: A Resettable Counter

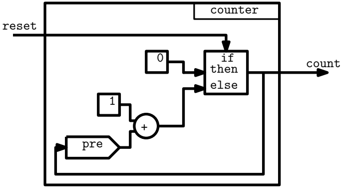
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset						...
count						...

Example: A Resettable Counter

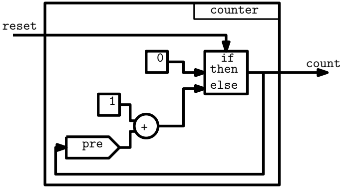
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False					...
count	0					...

Example: A Resettable Counter

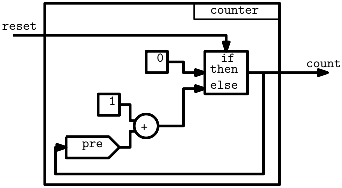
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False				...
count	0	1				...

Example: A Resettable Counter

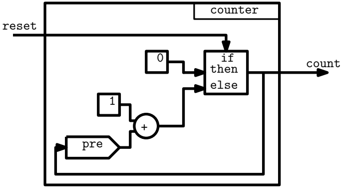
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False	False			...
count	0	1	2			...

Example: A Resettable Counter

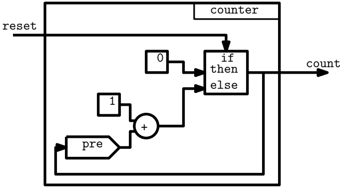
```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False	False	True		...
count	0	1	2	0		...

Example: A Resettable Counter

```
node counter(reset:bool)
     returns (count:int)
let
  count = 0 -> if reset
               then 0
               else pre(count)+1;
tel
```



time	0	1	2	3	4	
reset	False	False	False	True	False	...
count	0	1	2	0	1	...

Single assignment

Do not write:

```
node counter(reset:bool) returns (count:int)
let
  if reset then count=0 else count=0->pre(count)+1;
tel
```

▶ Each flow is only defined **once**.

Causality

Do not write:

```
node error(reset:bool) returns (count:int)
let
  x=y+1; y=x+2;
tel
```

▶ No immediate loop !

- Verified by a *causality analysis*;
- A `pre` is missing somewhere.

Clocks and undefined values

- Under-sampling (`when`) introduces undefined values, which must not be accessed. The flow is said to be *absent*.
- Only flows with the same clock can be combined: `x+x when c` is forbidden !

x	1	3	5	0	-2	...
c	False	False	True	False	True	...
current (x when c)						...

Trick :

- Use clocks initially true: `x when (true->c)`
- Force a default value

Plan

LUSTRE in the development cycle

- ✔ Write the synchronous program (formal, high abstraction level);
- **Compile** ▶generation of C code (medium abstraction level)

Valid Programs

Static analyses for checking correctness before the actual code generation:

- Causality (no cycle)
- Initialisation analysis (pre)
- **Clock calculus** ▶No access to absent values.

Clock calculus: type checking

Classical (ML-like) type inference/check: Main ideas :

- each expression has a type for values and another type for its clock;
- there is a type for the basic clock;
- a clock type is derived by applying operators on clocks : $+$ does not modify a clock type, but needs its two operands to be compatible; `pre` defines a subclock of its operands, `when` also.

►All expressions are typed. If not typable, the compiler rejects. [Equality of conditions is only syntactically checked]

Sequential Code Generation - Goal

Generate a (C) program of the form:

```
init(memory)
each period do
  read(inputs);
  outputs=f(M,inputs);
  memory=g(M,inputs)
  write(outputs)
done
```

step()

- Goal here : generate `init`, `f`, `g`, infinite loop

Simple syntax-based code generation

Each node is compiled into a separate procedure:

- flow definition ▶variable assignment;
- pointwise operator ▶classical operator;
- `pre`, `->` ▶memories;
- `when` ▶tests (if).
- sequentialization of the equation system.

Compilation of pre, example

```
node foo(i:int) returns (dec:int)
let
  dec = 0 -> i;
tel
```

compiles into (**new init variable**):

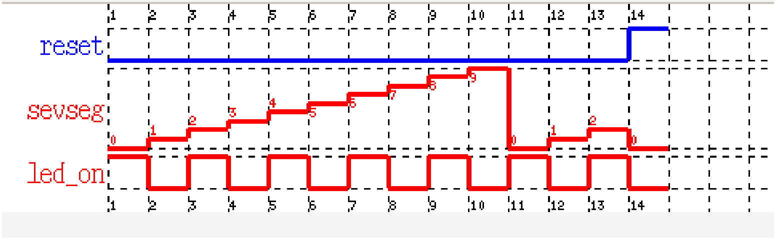
```
if init then dec = 0 else dec = y;
```

Compilation of our example as a simulation loop

```
node cpt(reset:bool) returns (led_on: bool) ;
let
   led_on = true -> not pre(led_on);
tel
```

```
lus2c demo_led.lus cpt -loop
```

- generates a .c and a main for simulation. (cf Lab)



Simulation loop : main

```
  while(1){
    // std input of data
   cpt_I_reset(ctx, _get_bool("reset"));
   // step, contains std outputs
   cpt_step(ctx);
}
```

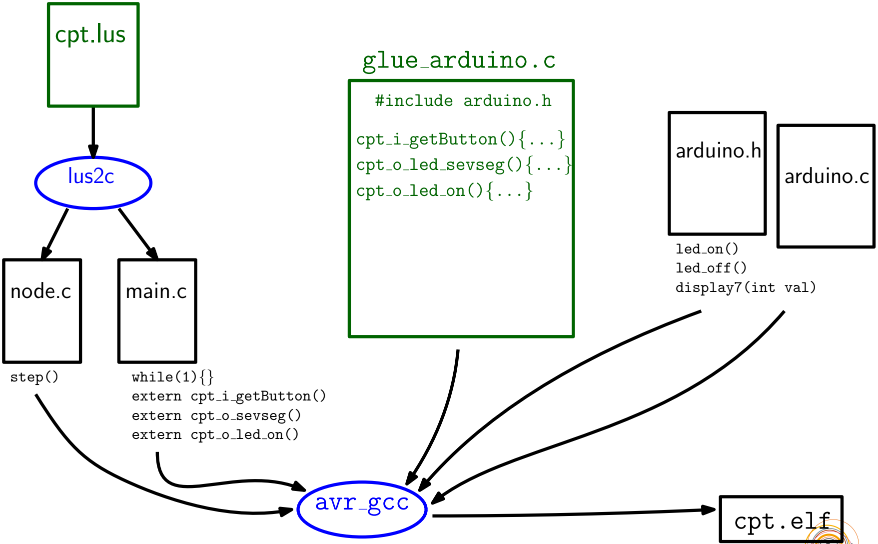
`ctx` is the state (defined in `cpt.c`).

What's next?

- ✔ Write the synchronous program (formal, high abstraction level);
- ✔ Compile : it generates C code (medium abstraction level);
- Write the **integration program**:
  - Read inputs on sensors;
  - Call the synchronous program;
  - Apply outputs to actuators.

▶Lab

Compilation chain for arduino backend



Plan

Formal verification, what for ?

- Reactive/real time systems are **critical**
- We want strong guarantees.

▶Both **functional** and timing properties.

Timing properties

Go and see WCET course. The analysis are all **sound**.

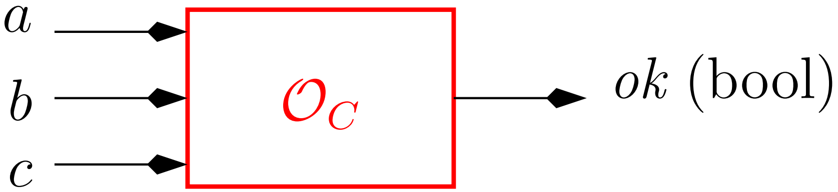
Functional properties, how?

Different approaches:

- Test the generated code on scenarios ▶not complete
- Formal verification of the generated code ?
- Formal verification of the source code ?
- Something in the middle ?

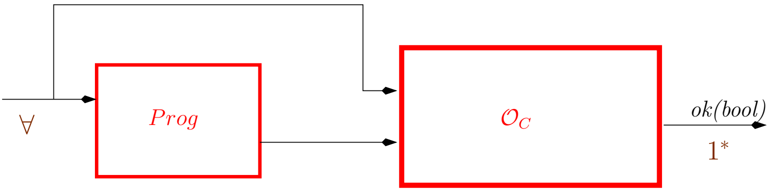
Lustre verification with observers, principle

- A property is specified in the same language as the program.
- The "execution" is done in parallel.



▶**Goal :** prove $L(\neg C) \cap L(Prog) = \emptyset$

General scheme of verification with observers



Model checking of the 20th century - demo

An example with `xlesar` and the node :

```
node edge (b : bool) returns (edge : bool);
let
    edge = false -> b and not pre b;
tel
```

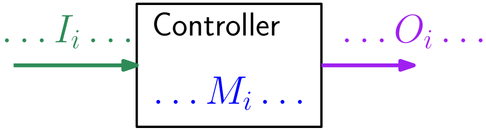
Let us try to prove :

- `true -> (edge => not pre edge)`
- `b => edge`

Recall : the Implicit State Machine

Synchronous programming model

- The **output function** $O_i = f(I_i, M_i)$;
- The **transition function** $M_{i+1} = g(I_i, M_i)$



This is equivalent to an explicit transition system.

Computing an explicit automaton from a lustre node

```
node edge (b : bool) returns (edge : bool);
let
    edge = false -> b and not pre b;
tel
```

By hand or `lus2atg`. Demo. (be careful, there seems to be a bug inside the atg viewer).

Computing an automaton from a lustre node+observer

```
node edge (b : bool) returns (edge : bool);
    let
       edge = false -> b and not pre b;
    tel

node obs (b:bool) returns (ok:bool);
var resu1:bool;
let
        resu1 = edge(b);
        ok = true -> (resu1 => not pre resu1);
tel
```

Demo : with luciole and with `lus2atg` (minimised automata).

Verifying boolean properties

Demo : model checker `lesar`. How ?

> *"It explores a finite model (an automaton) of the program. This model is an abstraction that represents an upper-approximation of all the possible executions of the program. The abstraction made on the program is conservative: if the verification suceeds on the model, the property is also satisfied by the program"*

If the program is purely boolean, the tool is correct and **complete**

Demo: edge2, cpt.

And for numeric properties ?

Some ideas :

- The Automaton is infinite in the general case.
- Its construction can be helped by the property to prove ($var \leq 2$ gives 2 states, ...)
- The enumerated forward strategy can be approximated (Abstract interpretation, here `nbac`).

Plan

Take-out message

For real-time:

- Programming reactive loops can be error-prone, we favor high level languages.
- Safety is important. Timing AND functional constraints.

▶Synchronous languages are one (among other) solution.

Perspectives in the CR10 (SEC) context

- Lustre is a DSL for reactive systems.
- Factorization : main infinite loop, only `init` and `step`.
- Nothing is magic (glue code).
- The language is designed with validation in mind. Its validation is at the same time generic and specific.

A DSL with many years of experience and relatively static (the perfect counterexemple).
[insert here discussion with Sebastien M]

Credits

Slides used for a common talk with Lionel Morel, INSA Lyon.

- *Julien FORGET (Cristal)* teaching notes
- *Abdoulaye GAMATIE (LIRMM)*, Synchronous Programming of Real-Time Systems with the Signal language, course at Telecom Lille 1, 2012;
- *Pascal RAYMOND (Verimag)*, various teaching notes.
- *Florence MARANINCHI (Verimag)*, Arduino inspiration.