Embedded Systems:

From High-Confidence Design

to Safe Execution

Lecture 2

Implementation of Synchronous Data-Flow Programs

Pascal Raymond

Verimag-CNRS

http://www-verimag.imag.fr/ raymond/

Université Grenoble Alpes

GRENOBLE INP UGA

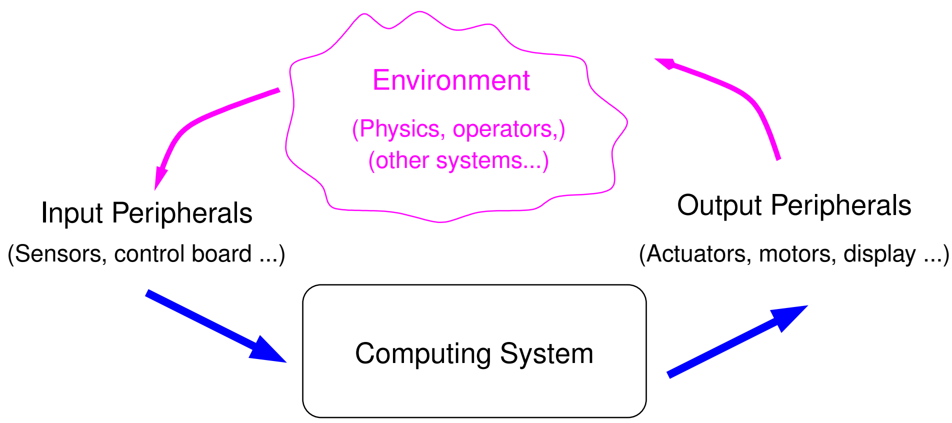
Summary

1. Towards safe embedded implementations

Embedded systems at work

Embedded systems...

- or *reactive / real-time / control engineering /...* systems
- Almost synonyms:
  - each term insists on a one characteristic
  - systems we are considering are all that
- The big picture:



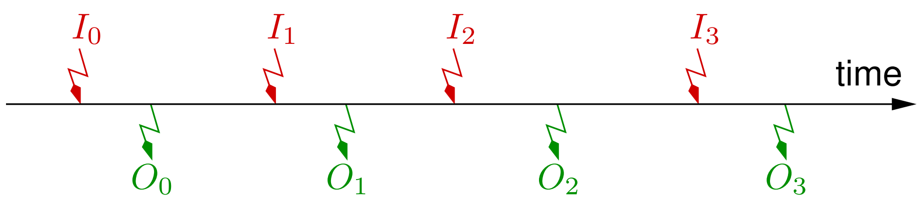
Implementation layers

- Hardware
- Firmware/OS
  - manage/access to peripherals
  - manage execution (tasks, real-time clocks)
- Software (application program, controller program)
  - perform a particular 'job'

Implementation safety

- functional: "computes the right outputs" (mainly a software problem)
- real-time: "computes fast enough" (involves ALL layers)

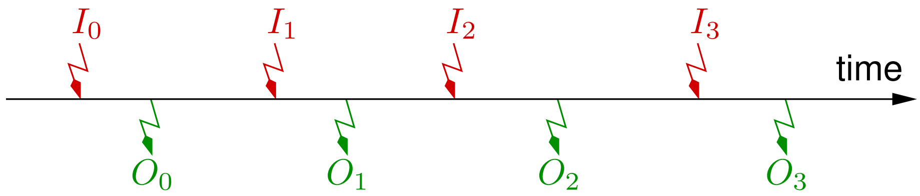
Observable behavior over time (virtually)



- sequence of Inputs/Outputs reactions
- system receives I_t and reacts by producing O_t, and so on...

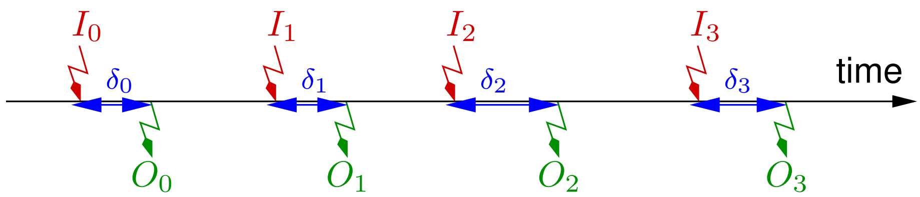
Is the system safe ?

Functional correctness



- Functionality: outputs O_t are the ”right” ones
  - mainly a software problem
  - depends on a particular application
  - at least, fundamental and generic property: determinism
    - a given sequence $I_0...I_t$ must always produce the same sequence $O_0...O_t$
- Note: synchronous languages (Scade/Lustre) are designed to guaranty by construction this property

Real-time correctness



- Real-time: the response delay δ_t is *short enough*
  - not universal: depends on the controlled environment
  - expected response deadlines range form 10ms to 50ms for physical world (transportation, energy)
  - from 100ms to several seconds for less critical systems (elevators, crane, weather station)
  - at least: the *worst case response time* (WCRT) must be known

Focus on functionality

- Determinism:
  - output O_t is determined by previous inputs,
  - i.e. it exists (conceptually) some a (mathematical) function Φ:

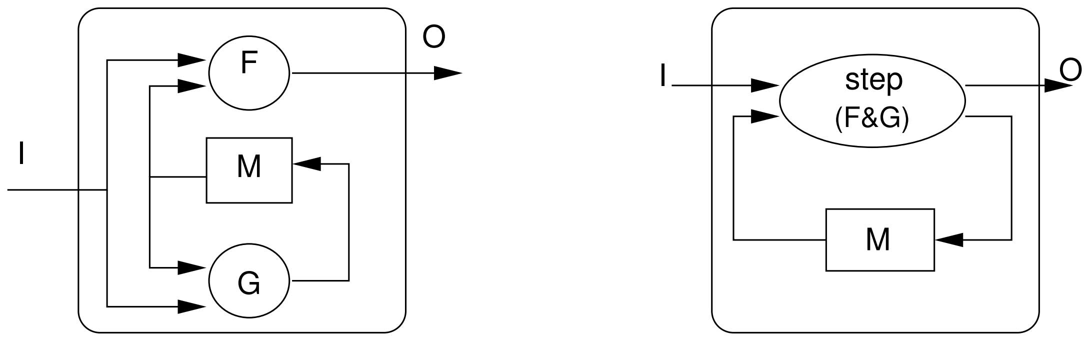
    $O_t = \Phi(I_0, \cdots, I_{t-1}, I_t)$
- Necessary memory MUST be bounded
  - otherwise existence of (finite) WCRET cannot be guaranteed
  - it exist a (finite) set of variables, M, with a given initial value M_0,
  - it exists a function F and a function G s.t.

    $O_t = F(M_t, I_t)$ (output function)

    $M_{t+1} = G(M_t, I_t)$ (transition, or state function)

Implementation principle

- concretely/in practice:
- F and G (the semantics) are implemented/computed jointly by a *transition* procedure (often called step procedure).



(conceptually) (more concretely)

- reactive behavior is implemented by calling the step procedure within a infinite loop.
- What about (infinite) main loop ?

Typical loop implementation: event-driven

```
init();
while(1) {
  wait_inputs();
  compute_step();
  emit_outputs();
}
```

- reaction triggered by some input event
- **wait_inputs()** and **emit_outputs()** are machine and OS dependent
- just a principle, concrete implementation depends on machine/OS

Typical implementation: time-driven (i.e. periodic)

```
init();
while(1) {
  wait_period();
  sample_inputs();
  compute_step();
  emit_outputs();
}
```

- reaction triggered by a periodic clock
- this is the choice for (almost) all critical embedded systems
- in this course: focus on this choice
- just a principle: may differ depending on machine/OS

Goal of in this course

- Sequential code generation
  - What synchronous languages compilers do (and do not do)
- Implementation of the main loop
  - with or without OS support
  - single task or multi-task

2. From data-flow to sequential code

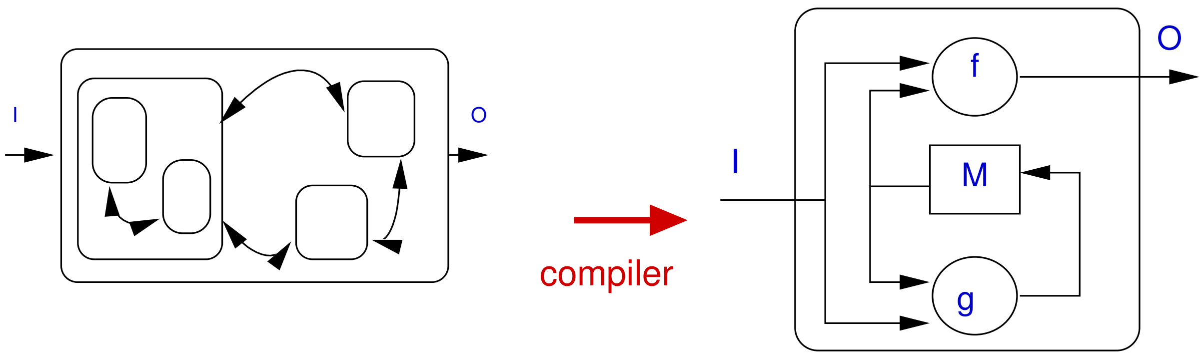
The (only) goal of synchronous compiler

- Synchronous languages compilers (SLC) are platform-agnostic:
  - do not target a particular hardware/firmware/OS
  - be as generic as possible
  - in particular do not generate binary (assembly) code:
    - all SLC generate C code
    - pragmatic: C is the *de facto* universal language for low-level programming, available for all platforms.
  - Only generate the *functional code* (init and step):
    - the loop code is too dependent on a particular hardware/OS

Compilation of synchronous programs

General problem

Transform a (hierarchic) parallel program into a (simple) sequential program.



Whole implementation of a reactive program P

```
var I, O, M;
M := m0; proc P_step() ...;
foreach step do
    read(I);
    P_step(); // combines: O := f(M, I); M := g(M, I);
    write(O);
end foreach
```

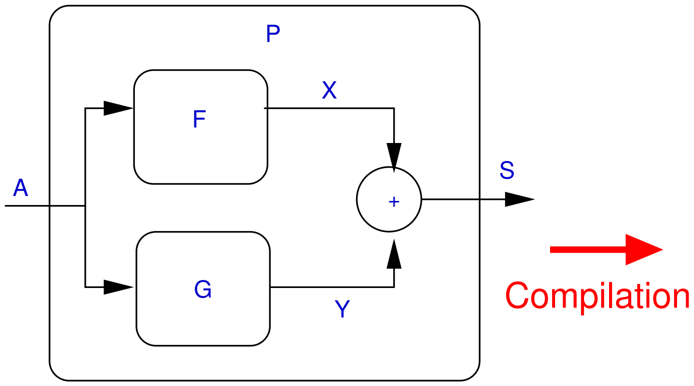
Job of the compiler

- Find the memory M and its initial value m0
- Build the core of the loop (the P_step procedure)
- As far as possible, generate efficient code

Modular compilation problem

The "obvious" way of compiling

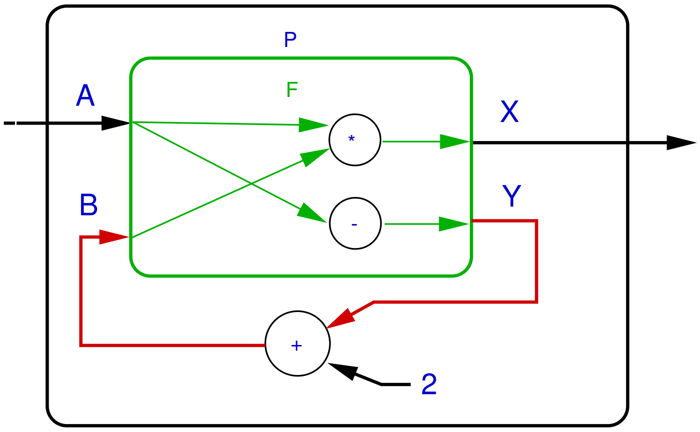
A Lustre node → a step procedure.




```
var A, X, Y, S;
proc F_step() begin X := ... end
proc G_step() begin Y := ... end
proc P_step()
begin
  F_step();
  G_step();
  S := X + Y;
end
```

Problem

What about feed-back loops ?



combinationnal loop ?

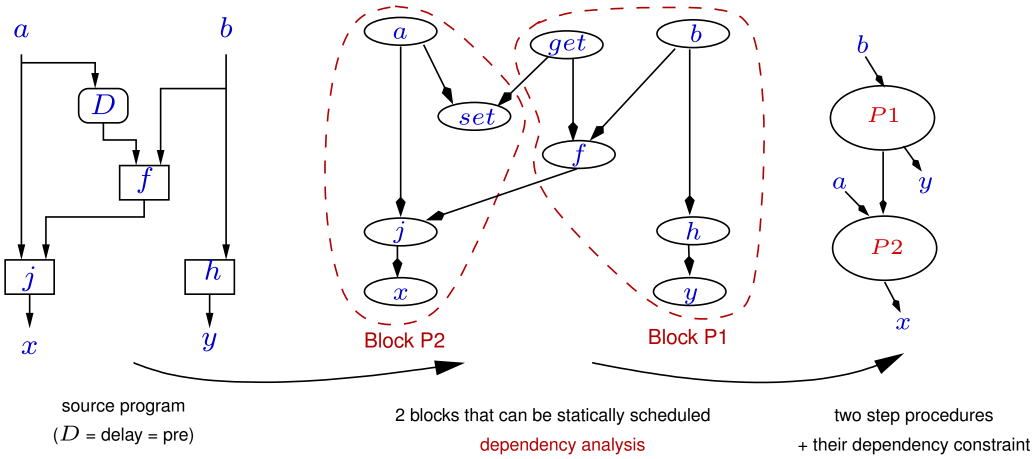
not when inlined ! ⇒
`X = A * (2 - A)`

- The program "is" correct (in a "parallel" world),
- but no `F_step` procedure can work!

Solution(s)

- Lustre (academic): expansion (i.e. inlining) of node calls
  - Strictly compliant with the principle of substitution.
  - Forbids modular compilation.
- Scade: feedback loops (without `pre`) are forbidden.
  - Reject correct parallel programs.
  - Allow modular compilation.
  - Reasonable choice in a industrial framework.
- Compilation into ordered blocks aka Modular Static Scheduling
  - Intermediate solution
  - Split the step into a minimal set of (sequential) blocks,
  - Only expand this simplified structure.

An example of Modular Static Scheduling



- Interesting theoretical result.
- Not (yet ?) used in industry.

Compilation of Lustre

Example : a filtered counter

- count rising edges of X (F),
- reset with a delay (R).

```
node CptF(X,reset: bool) returns (cpt: int);
var F, R : bool;
let
  cpt = if R then 0
    else if F then pre cpt + 1
    else pre cpt;
  R = true -> pre reset;
  F = X -> (X and not pre X);
tel
```

Simple loop compilation

Intuitively, do what is necessary to make definitions equivalent to assignments, i.e.:

- translate classical operators (trivial),
- replace `pre`'s and `->`'s with memory constructs,
- sequentialize according to data-dependencies (i.e. static scheduling).

Identify the memory

- Introduce a explicit variable for each **`pre`**:

```
pcpt = pre cpt;
preset = pre reset;
pX = pre X;
```

- Introduce a special memory

```
init = true -> false;
```

and replace each:

```
x -> y
```

with

```
if init then x else y
```

New version of the Lustre program

```
cpt = if R then 0
   else if F then pcpt + 1
   else pcpt;
R = if init then true else preset;
F = if init then X else (X and not pX);
pcpt = pre cpt;
preset = pre reset;
pX = pre X;
init = true -> false;
```


Sequentialization

Must take into account:

- Instantaneous dependences between values,
  - an (partial) order MUST exist (no combinational loop), example: `R` before `cpt` and `F` before `cpt`
  - chose a compatible complete order (schedule), example `R`, then `F` then `cpt`.
- Memorisations
  - Must be done at the end of the step, in any order.

Simple loop implementation (C-like code)

- Arithmetic and logic are translated "asit"
  (ex. **and** becomes **&&**, **if**..**then**..**else** becomes ..**?**..**:**..)
- `pre`'s are replaced with memories
- `->`'s are replaced with **init?**...**:**...
- Inputs/outputs are stores in global variables (for instance)

Simple loop implementation (C-like code)

```
int cpt; bool X, reset; /* I/O global vars */
int pcpt; bool pX, preset;  /* non initialized memories */
bool init = true; /* the only necessary initialization */
void CptFiltre_step() {
   bool R, F;/* local vars */
   R = init ?  true :  preset;
   F = init ?  X : (X && !  pX);
   cpt = R ? 0 :  F ? pcpt + 1 :  pcpt;
   pcpt = cpt; pX = X; preset = reset;
   init = false;
}
```

Optimizations

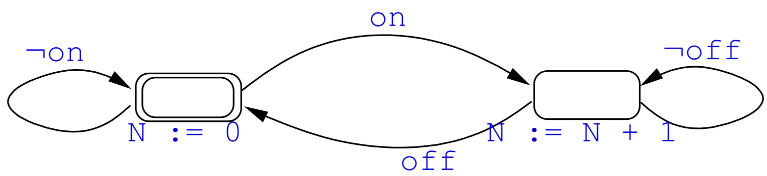
- Control structure: **?** becomes **if**
- Factorize conditions
- Eliminate useless local vars

```
if (init) {
    cpt = 0;
    init = false;
} else {
    F = (X && !  pX);
    cpt = preset ?  0 :  F ? (pcpt+1) :  pcpt;
}
pcpt = cpt; pX = X; preset = reset;
```


Compilation into automaton

Idea

The following reactive automaton:



is exactly equivalent to a Lustre program:

```
node Chrono (on, off : bool) returns (N : int);
var R : bool;
let
    R = false -> pre (if R then not off else on);
    N = if R then (pre N + 1) else 0;
tel
```

Problem: how to build the automaton from the Lustre code ?

Goal

- Automatically build an automaton equivalent to a Lustre program

  How ?

- Idea: an (explicit) state ⇔ a valuation of the memory
- N.B. finite number of states ⇒ finite memory (e.g Boolean)

Example of `CptF`

- **`S1`** = initial state = "**`init`** true, all other undefined"
- simplifed code : **`cpt = 0`**
- integer memorization: still the same
- Boolean memorization: state transition

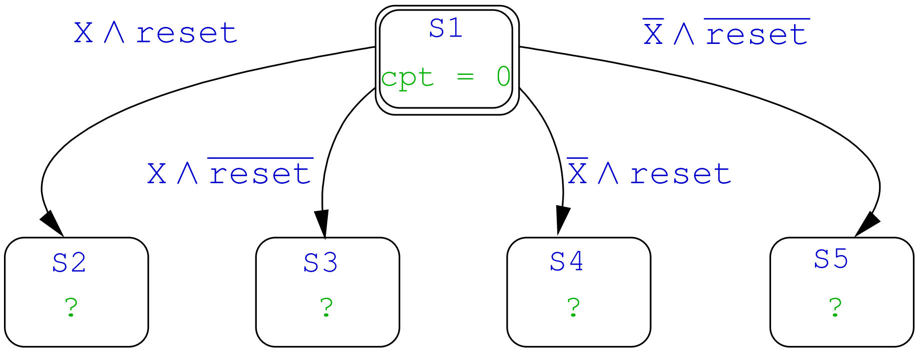
Transitions

- State **`S1`** (initial):

  `init = false; pX = X; preset = reset;`

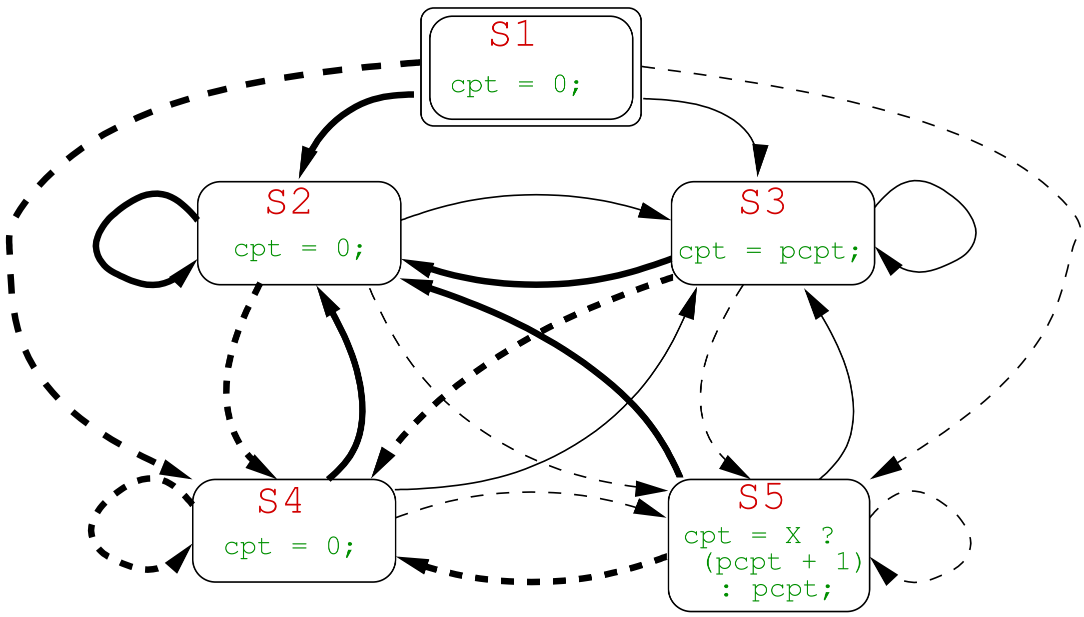
Depending on the values of `X` and `reset`, 4 next states:

- $\mathtt{X} \wedge \mathtt{reset} \rightarrow \mathtt{S2} \equiv \overline{\mathtt{init}} \wedge \mathtt{pX} \wedge \mathtt{preset}$
- $\mathtt{X} \wedge \overline{\mathtt{reset}} \rightarrow \mathtt{S3} \equiv \overline{\mathtt{init}} \wedge \mathtt{pX} \wedge \overline{\mathtt{preset}}$
- $\overline{\mathtt{X}} \wedge \mathtt{reset} \rightarrow \mathtt{S4} \equiv \overline{\mathtt{init}} \wedge \overline{\mathtt{pX}} \wedge \mathtt{preset}$
- $\overline{\mathtt{X}} \wedge \overline{\mathtt{reset}} \rightarrow \mathtt{S5} \equiv \overline{\mathtt{init}} \wedge \overline{\mathtt{pX}} \wedge \overline{\mathtt{preset}}$



- Code of the other states:
  - `S2` → `cpt = 0`
  - `S3` → `cpt = pcpt`
  - `S4` → `cpt = 0`
  - `S5` → `F = X`, `cpt = X? (pcpt + 1) : pcpt`
- Transitions of the other states:
  - same than `S1` (only depend on inputs)

Finally ...



⇒ problem: size!

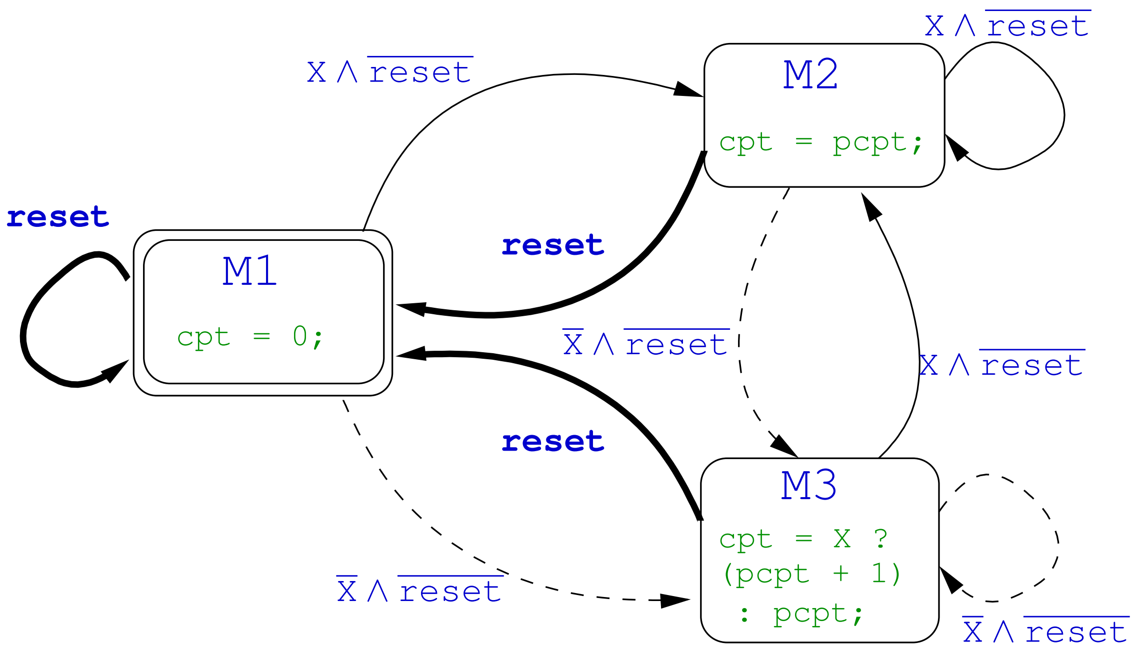
Remarks on the size

- n memories ⇔ (worst case) 2^n states, 2^{2n} transitions
  ⇒ *Combinatorial explosion*

But not always:

- Unreachable states
  - Example : `(pre X, pre(X or Y))` ⇒ "only" 3 states
  - Counter-example : `CPtF` !
- State equivalence
  - Example `CPtF` : `S1`, `S2` et `S4` "*are doing the same thing*"
    ⇒ *Importance of producing a minimal automaton*

Minimal automaton of `CptF`



Implementation en C

With a switch (for instance):

```
typedef enum {M1, M2, M3} TState;
TState state = M1;
void CptFiltre_step(){
   switch (state) {
      case M1:  cpt = 0; break;
      case M2:  cpt = pcpt; break;
      case M3:  cpt = X? (pcpt + 1):pcpt; break;
   }
   pcpt = cpt;
   if (reset) state = M1;
   else if (X) state = M2;
   else state = M3;
}
```


Simple loop or automaton ?

- Automaton
  - Optimal in computation time
  - Possibly huge size
- Simple loop
  - Slightly slower
  - Linear size

  ⇒ Only *reasonable solution* in industry
- Interest of Automata
  - Not satisfactory for code generation, but ...
  - Precious for *reasoning* about programs, i.e. for validation/verification

C-code interface

- The compiler must provide a standard API for the sequential code, with precise convention for:
  - the name of the generated procedures
  - the way internal memory is allocated and accessed
  - the way input/output parameters are given/retrieved
- Plenty of solutions and variants, depend on the compiler and its options

Example: Scade-kcg generated header

Scade profile:

```
node FOO(Ga: bool; Bu: int) returns (Zo: int; Meu: real);
```

kcg

generates `foo.c` and the corresponding header file `foo.h`:

```
#include "kcg_types.h"
//==== context type ===========
typedef struct {
  //---- outputs -------------
  kcg_int Zo;
  kcg_real Meu;
  //----- locals ---------------
...
} outC_FOO;
//=== node initialization and cycle
extern void FOO(kcg_bool Ga, kcg_int Bu, outC_FOO *outC);
extern void reset_FOO(outC_FOO *outC);
```

Example: Scade-kcg conventions (cntd)

- Outputs and local memory are stored in a single structured type (the context)
  Allocation of the structure is up to the user (in glogal memory, head, stack)
- to initialize the context, a *reset* procedure is provided, that takes as input a pointer to the context,
- the step procedure:
  - takes the list of input parameters (by value),
  - a pointer on the context,
  - and returns nothing
- after a step call, the user can retrieve the outputs values stored in the context
- N.b. the compiler does not fix the implementation of basic types:
  user has to define them in `kcg_types.h`
- Very similar solution adopted for other Lustre-like compilers (Lustre V6, octogon, velus)

Example: Lustre/lus2c conventions

```
#include "FOO_ext.h"
//-- Context type (abstract)
struct FOO_ctx;
//-- Context allocation
extern struct FOO_ctx* FOO_new_ctx(void* client_data);
//-- Input procedures:
// provided, must be called before each 'step'
extern void FOO_I_Ga(struct FOO_ctx* ctx, _boolean);
extern void FOO_I_Bu(struct FOO_ctx* ctx, _integer);
//-- Output procedures:
// not provided, must be defined by the user
//void FOO_O_Zo(void* cdata, _integer);
//void FOO_O_Meu(void* cdata, _real);
//-- Reset procedure
extern void FOO_reset(struct FOO_ctx* ctx);
//-- Step procedure
extern void FOO_step(struct FOO_ctx* ctx);
```

Example: Lustre/lus2c conventions (cntd)

- Clearly inspired by OO (Object Oriented) domain
  - The code is an (incomplete) class:
    - new, step and reset ”methods”
    - inputs methods
    - ”virtual/undefined” output methods
  - The user must complete/derive its own class:
    - add (if needed) its own data/variables (client-data mechanism)
    - define the output method
  - Very general and versatile...
- Simplified conventions
  - Works when a single node instance is needed
  - No need for ”new” and the client-data mechanism (heap-free)
  - A single context is statically allocated (and hidden to the user)
  - Sufficient for this course
  - Concretly `-ctx-static` option

Example: lus2c with static context conventions

```
#include "FOO_ext.h"
//-- Input procedures:
// provided, must be called before each 'step'
extern void FOO_I_Ga(_boolean);
extern void FOO_I_Bu(_integer);
//-- Output procedures:
// not provided, must be defined by the user
//void FOO_O_Zo(_integer);
//void FOO_O_Meu(_real);
//-- Reset procedure
extern void FOO_reset();
//-- Step procedure
extern void FOO_step();
```

3. Real-time implementation

Implementation platform

How to run a (periodic) RT application ?

- strongly depends on platform, not universal...
- ...however, embedded systems platform provides similar features

The right questions when discovering a platform

- How to access the peripherals (read inputs, write outputs) ?
- How to achieve periodicity (i.e. real-time support) ?
- How to compile/upload/run my application ?

Example platform: Arduino+BatCar

Arduino

- formally: a micro-controller
- tiny, simple, (cheap!), designed for teaching purpose
- representative, not so different from more industrial boards (e.g. Freescale NXP)
- processor is a 16bits Atmel/AVR
- provides generic input/output ports
- each port must be programmed depending on the actual peripheral
- programming language is C++
- Arduino firmware consists of a generic reactive program:
  - basically a sequence of initializations, followed by an infinite loop
  - with 2 'hooks' (functions that must be provided by the user):
    - `setup()` where to put user initializations
    - `loop()` the core of the infinite loop

BatCar

- Arduino + a set of peripherals
- Inputs:
  - a button (called k1, Boolean)
  - 2 light sensors (left and right, Boolean)
- Outputs:
  - 2 motors (left and right, integer)
  - a buzzer (Boolean)
  - 3 leds (red, yellow, green, Boolean)
- Interface between peripherals and Arduino ports is a little bit technical
  we use an (existing) API with straightforward features, e.g.:

```
BatCar.init_button();
BatCar.set_motor_left(int);
```

  etc.

The Lustre part

- Suppose we have developped a BatCar controller in Lustre, whose profile is:

```
node control(
  k1: bool; sensor_left, sensor_right: bool
) returns (
  motor_left, motor_right: int;
  red_light, yellow_light, green_light: bool;
  buzzer: bool
);
```

- Lustre compiler generates a code defining:

```
void control_reset();
void control_step();
void control_I_k1(bool);
void control_I_sensor_left(bool);
void control_I_sensor_left(bool);
```

- and expecting the definition of output functions, e.g.

```
void control_O_motor_left(int);
void control_O_red_light(bool);
```

etc.

Programming the reactive glue

- Output functions calls the BatCar API, e.g.

```
void control_O_motor_left(int v){
  BatCar.set_right_speed(v);
}
void control_O_buzzer(bool v){
  BatCar.set_buzzer(v);
}
```

etc.

- Arduino's user setup must contain BatCar and Lustre init

```
void setup(){
  BatCar.init_button();
  BatCar.init_line_sensors();
  BatCar.init_motors();
  BatCar.init_buzzer();
  control_reset();
}
```

Programming the reactive glue (cntd)

- Arduino's user loop must contain input sampling and lustre step

```
void loop(){
  control_I_k1(BatCar.button_pressed());
  control_I_sensor_left(BatCar.line_sensor_left());
  control_I_sensor_righ(BatCar.line_sensor_right());
  control_step();
}
```

- Warning: not real-time periodic ! loops as far as possible

Basic RT support in Arduino

- Arduino provides a hardware clock, accessed via the functions:

```
unsigned long millis(); //current time in ms
void delay(unsigned long d); //spend d ms doing nothing
```


Programming the reactive glue (cntd)

- Arduino's user loop with RT periodic 'wrapper'

```
#define PERIOD 30
void loop(){
  unsigned long t0 = millis();
  control_I_k1(BatCar.button_pressed());
  control_I_sensor_left(BatCar.line_sensor_left());
  control_I_sensor_righ(BatCar.line_sensor_right());
  control_step();
  unsigned long t1 = millis();
  delay(PERIOD-(t1-t0));
}
```

- N.B. RT achieved by polling (active waiting)

Using a Real-Time OS

What for ?

- main characteristic: multi-tasking, preemptive scheduling
- with a precise notion of system clock (periodic)
- not (really) necessary for single task appli...
- ... however let see how it works

RTOS features

- Several RTOS, each with their own API
- Same principles (task creation, wait/sleep on real-time clock, start scheduling)
- Example: FreeRTOS

FreeRTOS API

- Reference **`https://www.freertos.org/`** + Kernel/API Reference
- Create a task (see `xTaskCreate`):
  - to be done at initialization
  - args are: code to execute (procedure), priority, user data etc.
- Start the scheduller (see `vTaskStartScheduler`)
  - to be called when all tasks are created
  - no argument, never returns
- Real-time support (see `vTaskDelayUntil`)
  - to be called within the task code
  - forces the task to 'sleep' for a precisely timed delay
  - N.b. time is counted in system ticks
  - default: 1 system tick = 15 ms
- We'll try it in the practical work

Multi-tasking

Multi-tasking, safety and real-time

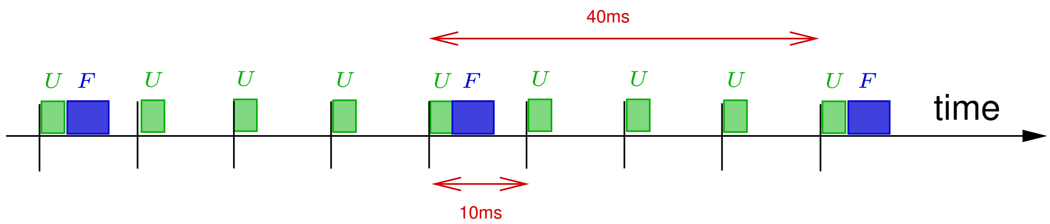
- Basically: (dynamic) multi-tasking is bad for safety and real-time
  - hard to guarantee real-time (blocking, starving ...)
  - hard to guarantee safety (non-determinism, priority inversion ...)
- But it may be interesting (even necessary) in (at least) one case:
  - a (slow) task must compute less often than others

Non-preemptive multi-tasking

- Example: U must compute each 10ms, F each 40ms
- This can be done in synchronous languages (Scade/Lustre):
  - U computes all the time, F computes 1 of 4 time
  - can be programmed with basic language, or using 'clocks' (out of scope)

ke

- At execution:

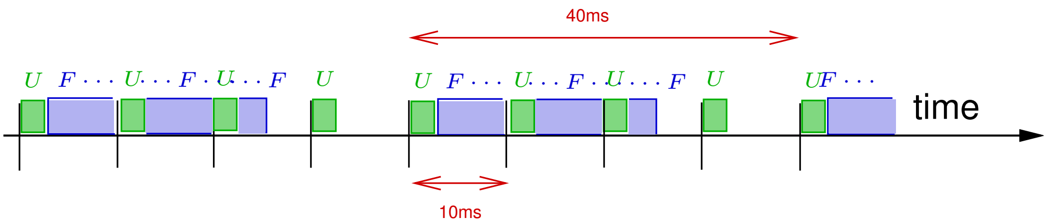


  - F computes less often, bu must compute 'fast'
  - WCET = WCET(U) + WCET(F)

Preemptive multi-tasking required

- A task (F) must be executed 'less often' than a task (U)

  because it takes more time to execute
- Example:
  - U executes each 10ms, with WCET(U) = 3ms
  - F executes each 40ms, with WCET(U) = 15ms



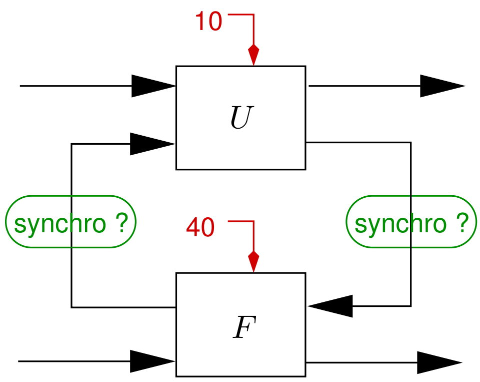
- Classical schedulability problem:

  $$1 \times WCET(F) + 4 \times WCET(U) = 27 < 40ms$$

  real-time is guaranted

Communication and determinism

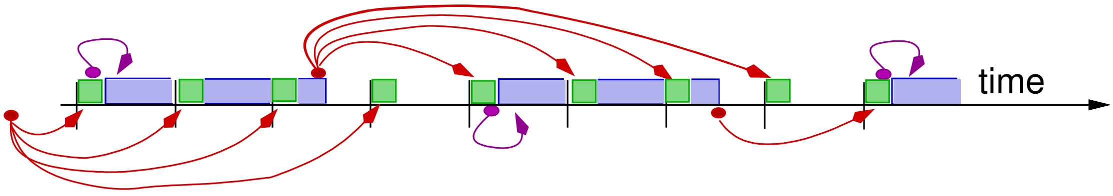
- General communication case:



- possible synchro:
  - (none) = freshest value, may work but not deterministic (depends on priority and actual computation time)
  - logical delay = strictly past value on the corresponding clock (e.g. F to U: take the value at the previous 40ms tick)

Deterministic scheme

- Mixed (deterministic) solution:
  - Short task has priority (U = Urgent)
  - Long task reads freshest value
  - Short task reads delayed value



- A little bit technical/costly to implement (double-buffering)
- Freshest-value principle is often accepted (relaxed determinism)