

Génération de code séquentiel pour Lustre

Marc Pouzet

`Marc.Pouzet@ens.fr`

Calcul parallèle et réactif

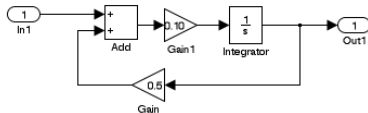
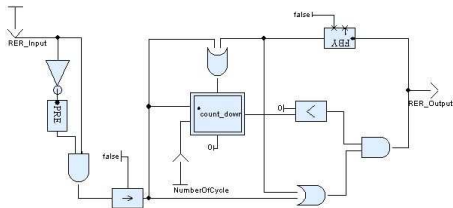
M1

Octobre 2020

The problem

- **Input:** a **parallel** data-flow network made of synchronous operators.
E.g., **Lustre**, **Scade**, **Simulink**
- **Output:** a sequential procedure (e.g., C, Java) to compute one step of the network: **static scheduling**

Examples: **Scade** and **Simulink**



This is part of a more general question

How to “compile the parallelism”, i.e., generate seq. code which:

- preserves the parallel semantics,
- treats all programs with no ad-hoc restriction.

Why sequentializing a parallel program?

- Often far more efficient than the parallel version.
- Get a **time predictable implementation** (real-time system).
- At the moment, tools for analysing the Worst Case Execution Time (WCET) work well for sequential code only.

This is not contradictory with the question of generating parallel code.

Both questions are interesting.

The basic intuition

Implement $f : \text{Stream}(T) \rightarrow \text{Stream}(T')$ as a pair (s_0, f_t) :

- an **initial state** $s_0 : S$;
- a sequential **step function**: $\langle f_t : S \times T \rightarrow T' \times S \rangle$

An equation $y = f(x)$ can be computed **sequentially** such that:

$$\forall n \in \mathbb{N}, y_n, s_{n+1} = f_t(s_n, x_n)$$

In practice, the internal state is modified **in place**.

Equivalently, decompose the step function in two:

- an **initial state**: $s_0 : S$
- a **value function**: $f_v : S \times T \rightarrow T'$
- a **transition function** (“commit”): $f_c : S \times T \rightarrow S'$

$$\forall n \in \mathbb{N}, y_n = f_v(s_n, x_n) \wedge s_{n+1} = f_c(s_n, x_n)$$

The problem is more complex when f has several inputs/outputs.

Two classical implementations

- Periodic sampling

```
s := s0;  
every clock tick  
  read_input e;  
  let o, s' = ft s e in  
  s := s';  
  write_output o  
end
```

- Event driven

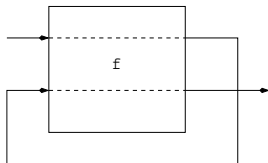
```
s := s0;  
everytime e is present  
  let o, s' = ft s e in  
  s := s';  
  write_output o  
end
```

Check that the inter-arrival time between events or clock ticks is greater than the WCET of the read/compute/write.

Modular Static Scheduling

Sequentializing parallel code cannot be done once for all, independently from the context.

Example ¹



```
node copy(a, b:bool) returns (c, d:bool);  
  let  
    c = a; (* 1 *)  
    d = b; (* 2 *)  
  tel;  
  
node loop(t:bool) returns (z:bool);  
  var y: bool;  
  let (y, z) = copy(t, y);  
  tel;
```

`loop(t)` would run perfectly in a parallel implementation.

¹The example is due to Georges Gonthier.

Modular Static Scheduling

```
void step_copy_one(int *a, int *b, int *c, int *d) {  
    *c = *a;  
    *d = *b;}
```

```
void step_copy_two(int *a, int *b, int *c, int *d) {  
    *d = *b;  
    *c = *a;}
```

```
void loop_step(int *t, int *z) {  
    int y;  
    step_copy_one(t, &y, &y, z);}
```

Only the first implementation of copy can be used.

Generating a **single** transition function is not sufficient.

Two main approaches has been followed in synchronous compilers.

The two main approaches to code generation

Maximal Static Expansion (“white boxing”)

- Function calls are statically (inlined).
- The way it is done in the **Lustre** compiler (VERIMAG).
- Efficient enumeration techniques can be applied to generate finite state automata [Raymond PhD. Thesis[11], Halbwachs et al. [7]].
- But code size can be prohibitive.

Single Loop Code Generation (“black boxing”)

- A **single code** repeated infinitely.
- Modular: **one node** produces **one step function**.
- Makes **tracability** of the compiler simpler.
- Imposes **stronger causality constraints**: every loop must cross a delay.
- The approach of **Scade** KCG.

An intermediate solution (“grey boxing”)

- Instead of producing a single step function per node, produce several together with a partial order.
- They must be called in an order compatible with this partial order.
- E.g., for copy, produces two, one that computes $c := a$, one that computes $d := b$.²
- This is called the **Modular Static Scheduling** problem.
- Identified in 1988 by Pascal Raymond who proposed a first algorithm. [10]
- The **Optimal Modular Static Scheduling** problem is when the number of step functions is minimal.
- Several solutions has been proposed. See extra course.

In this class, we focus on the simpler **single loop code generation** problem.

²Surely, for such a small function, inlining is preferable!

Single loop code generation

A reference compiler for a synchronous data-flow kernel ³

MiniLS: a minimalistic **clocked data-flow language** as input.

- used as some sort of “typed assembly language”.
- General enough to be used as a target language for **Lustre**.
- and a target for control structures like hierarchical automata.

Objective

- Single loop code generation.
- Compilation into an intermediate “object based” language that represent transition functions.
- Then, translated into imperative code (e.g., structured C, OCaml, Java).

³The notes are adapted from [1]

Organization of the Compiler



The source language

Expressions:

$$\begin{aligned} a ::= & v \mid x \mid v \text{ fby } a \\ & \mid op(a, \dots, a) \\ & \mid a \text{ when } C(x) \\ & \mid \text{merge } x \ (C \rightarrow a) \dots (C \rightarrow a) \end{aligned}$$

Equations:

$$D ::= x = a \mid (x, \dots, x) = f(a, \dots, a) \text{ every } a \mid D \text{ and } D$$

Function definitions, constants:

$$d ::= \text{node } f(p) = p \text{ with var } p \text{ in } D$$

$$p ::= x : bt; \dots; x : bt$$

$$v ::= C \mid i$$

When/Merge

h	true	false	true	false	...
x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
$v \text{ fby } x$	v	x_0	x_1	x_2	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$...
$z = x \text{ when true}(h)$	x_0		x_2		...
$t = y \text{ when false}(h)$		y_1		y_3	...
$\text{merge } h$ $(\text{true} \rightarrow z)$ $(\text{false} \rightarrow t)$	x_0	y_1	x_2	y_3	...

- $v \text{ fby } x$ is the unit delay initialized with v .
- the merge constructs combines two complementary sequences

Example (counter)

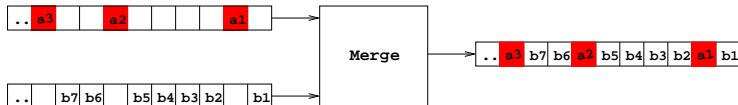
“Counts the number of occurrences of tick”.

```
node counting (tick:bool) = (o:int) with
var v: int in
  o = (0 fby o) + v
and v = if tick then 1 else 0
```

tick	true	true	true	false	false	true	true	...
o	1	2	3	3	3	4	5	...
v	1	1	1	0	0	1	1	...

The n-ary merge operator

- The merge $c \times y$ operator combines two complementary flows (flows on complementary clocks) to produce a faster one



Example: `merge c (a when c) (b whennot c)`

Generalization:

- generalized to n inputs of an enumerated type t with:

$$t = C_1 \mid \dots \mid C_n$$

- the sampling `e when c` is now written `e when true(c)`, i.e.,

$$bool = true \mid false$$

Resetting a behavior

How to make the node counting “resetable”, that is, whenever r is true, all its internal registers are reset to an initial value?

One has to reprogram counting into:

```
node counting_r (r:bool; tick:bool) = (o:int) with
  var v: int in
    o = (if r then 0 else 0 fby o) + v
    and v = if tick then 1 else 0
```

- There is no modular **reset** in Lustre. Making a component “resetable” is painful and error prone.
- The above encoding would lead to very bad sequential code.
- Two good reasons to make it a primitive in the language.

Specific notation:

$$f(a_1, \dots, a_n) \text{ every } c$$

all the registers used in the definition of node f are reset when the boolean condition c is true

Derived Operators

Mux/conditional:

$$\begin{aligned} \text{if } x \text{ then } e_2 \text{ else } e_3 &= \text{merge } x \\ &\quad (\text{true} \rightarrow e_2 \text{ when true}(x)) \\ &\quad (\text{false} \rightarrow e_3 \text{ when false}(x)) \end{aligned}$$

Initialization and un-initialized delay:

$$y = e_1 \rightarrow e_2 = y = \text{if } \textit{init} \text{ then } e_1 \text{ else } e_2$$

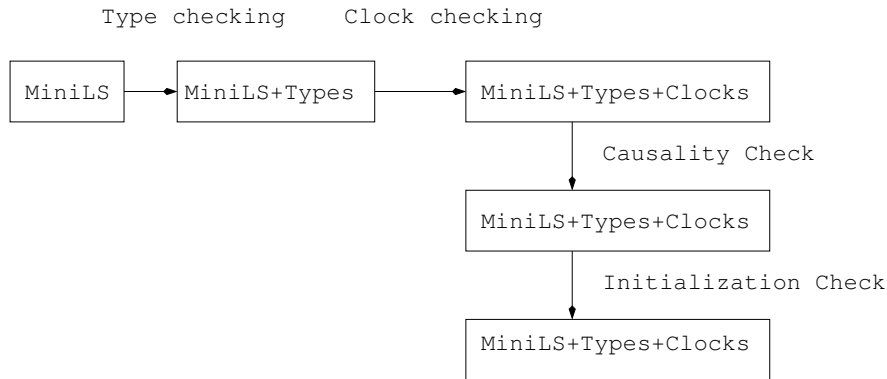
and $\textit{init} = \text{true fby false}$

$$\text{pre}(e) = \textit{nil} \text{ fby } e$$

where \textit{nil} is a value of the type of e

Either add them to the language kernel or express them into it.

Static Checking



Type and Clock checking

An intermediate language where every expression is **annotated** with a type expression (*bt*) and a clock expression (*ck*).

$$\begin{aligned} a &::= e_{bt}^{ck} \\ e &::= v \mid x \mid v \text{ fby } a \mid a \text{ when } C(x) \mid op(a, \dots, a) \\ &\quad \mid \text{merge } x (C \rightarrow a) \dots (C \rightarrow a) \\ D &::= x = a \mid f(a, \dots, a) \text{ every } a \mid D \text{ and } D \\ d &::= \text{node } f(p) = p \text{ with var } p \text{ in } D \end{aligned}$$

Clock expression:

- The clock $clock(s)$ of a sequence s is a boolean sequence such that $clock(s) = true$ iff s is present
- An expression e is annotated a boolean formula ck

$$ck ::= \text{base} \mid ck \text{ on } C(x)$$

- base is the “base” clock of the node; it is always true.
- $ck \text{ on } C(x)$ is true when ck is true and $x = C$.

Clock Verification

Type and clock checking are performed in order.

- Clock environment: $\mathcal{H} ::= [ck_1/x_1; \dots; ck_n/x_n]$
- $\mathcal{H} \vdash a : ck$ means that a is well annotated in \mathcal{H} with clock type ck .

$$\begin{array}{c} \text{(ANNOT)} \\ \mathcal{H} \vdash e : ck \\ \hline \mathcal{H} \vdash e_{bt}^{ck} : ck \end{array}$$

$$\begin{array}{c} \text{(OP)} \\ \mathcal{H} \vdash a_1 : ck \dots \mathcal{H} \vdash a_n : ck \\ \hline \mathcal{H} \vdash op(a_1, \dots, a_n) : ck \end{array}$$

$$\begin{array}{c} \text{(CONST)} \\ \mathcal{H} \vdash v : \text{base} \end{array}$$

$$\begin{array}{c} \text{(VAR)} \\ \mathcal{H}, x : ck \vdash x : ck \end{array}$$

$$\begin{array}{c} \text{(FBY)} \\ \mathcal{H} \vdash a : ck \\ \hline \mathcal{H} \vdash v \text{ fby } a : ck \end{array}$$

$$\begin{array}{c} \text{(WHEN)} \\ \mathcal{H} \vdash a : ck \quad \mathcal{H} \vdash x : ck \\ \hline \mathcal{H} \vdash a \text{ when } C(x) : ck \text{ on } C(x) \end{array}$$

$$\begin{array}{c} \text{(MERGE)} \\ \mathcal{H} \vdash x : ck \quad \mathcal{H} \vdash a_1 : ck \text{ on } C_1(x) \dots \mathcal{H} \vdash a_n : ck \text{ on } C_n(x) \\ \hline \mathcal{H} \vdash \text{merge } x (C_1 \rightarrow a_1) \dots (C_n \rightarrow a_n) : ck \end{array}$$

Clock Checking

We consider the simple case where all input/outputs of a node have the same clock ⁴

$$\begin{array}{c} \text{(CALL)} \\ \hline \forall i \in [1..k] \mathcal{H} \vdash x_i : ck \quad \forall i \in [1..n] \mathcal{H} \vdash a_i : ck \quad \mathcal{H} \vdash c : ck \\ \hline \mathcal{H} \vdash (x_1, \dots, x_k) = f(a_1, \dots, a_n) \text{ every } c : ck \end{array}$$

$$\begin{array}{c} \text{(NODE)} \\ \hline \vdash_{\text{base}} p : \mathcal{H}_p \quad \vdash_{\text{base}} q : \mathcal{H}_q \quad \vdash r : \mathcal{H}_r \quad \mathcal{H}_p, \mathcal{H}_q, \mathcal{H}_r \vdash D \\ \hline \vdash \text{node } f(p)(q) = \text{var } r \text{ in } D \end{array}$$

$$\begin{array}{c} \text{(PAT)} \\ \vdash x_1 : bt_1, \dots, x_n : bt_n : [x_1 : ck_1; \dots; x_n : ck_n] \end{array}$$

$$\begin{array}{c} \text{(PARAM)} \\ \vdash_{\text{base}} x_1 : t_1, \dots, x_n : t_n : [x_1 : \text{base}; \dots; x_n : \text{base}] \end{array}$$

⁴Lustre is more powerful as it allows for input/outputs to be on different clocks provided the first input is on the base clock.

Clock checking

$$\frac{\text{(EQ)} \quad \mathcal{H} \vdash x : ck \quad \mathcal{H} \vdash a : ck}{H \vdash x = a}$$

$$\frac{\text{(AND)} \quad \mathcal{H} \vdash D_1 \quad \mathcal{H} \vdash D_2}{\mathcal{H} \vdash D_1 \text{ and } D_2}$$

This system is very limited.

The language kernel has no data-structures, no type polymorphism.

All inputs and outputs must be on the same clock.

Extensions

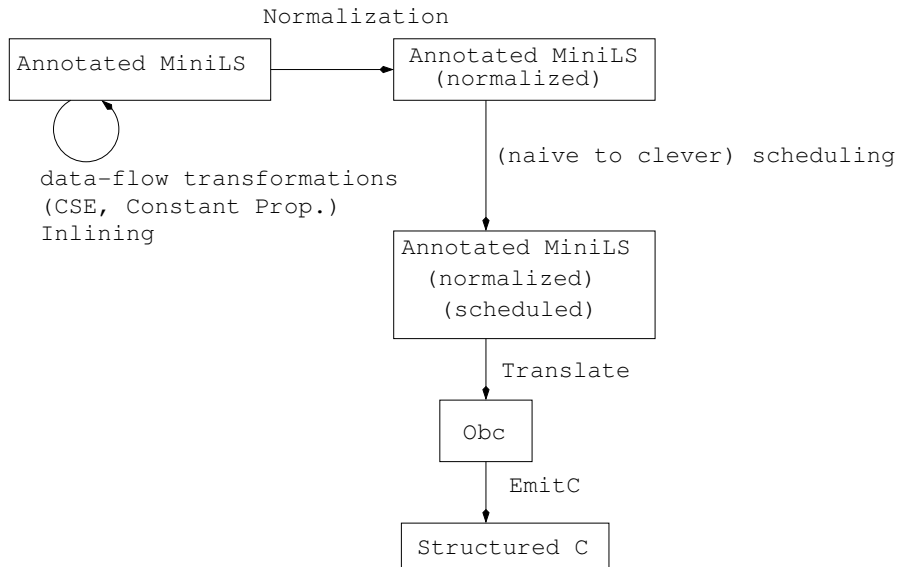
Some examples of equations and functions that can be defined in **Scade 6**.

```
x = (1, 2)
x = {a = 1; b = 2}
x = if c then (1,2) else (3, 4)
```

```
node hold(i: 't; clock h: bool; a: 't when h) returns (o:'t)
  o = merge(h; a; (i -> pre o) when not h);
```

- Enrich the type language and clock type language with tuples and polymorphism.
- The clock calculus of Lustre can be defined as a dependent type system [4, 2]
- A simpler one reminiscent of the ML-type system [5].
- Programs from the enriched language can be translated into the basic language.
- Yet, it is sufficient to illustrate how clocks are used to generate code.

Translation into sequential code



Putting Equations in Normal Form

- Prepare equations before the translation.
- Identify state variables vs temporaries.
- Rewrite equations such that delays and function applications do not appear in nested expressions.

Normal Form:

$$a ::= e_{bt}^{ck}$$

$$e ::= a \text{ when } C(x) \mid op(a, \dots, a) \mid x \mid v$$

$$ce ::= \text{merge } x \ (C \rightarrow ca) \dots (C \rightarrow ca) \mid e$$

$$ca ::= ce_{bt}^{ck}$$

$$eq ::= x = ca \mid x = v \text{ fby } a \\ \mid (x, \dots, x) = f(a, \dots, a) \text{ every } a$$

$$D ::= eq \mid eq \text{ and } D$$

Notation: $[eq_1; \dots; eq_n]$ or $(eq_i)_{i \in [1 \dots n]}$ for eq_1 and $\dots eq_n$. $eq_1.eq_2$ for the concatenation.

Checking the correctness of the normalization

The normalization is a simple rewritting. Its correctness can be validated independently, a posteriori.

- Given a list of equations, a (non trusted) normalization function returns a list of normalized equations and a substitution:

$$\text{normalize}([eq_1; \dots; eq_n]) \stackrel{\text{def}}{=} [eq'_1; \dots; eq'_m], [e_1/x_1, \dots, e_k/x_k]$$

- Check that:

$$\text{sub}([eq'_1; \dots; eq'_m])[e_1/x_1, \dots, e_l/x_k] = [eq_1; \dots; eq_n] = eqs$$

with:

$$\text{sub}(eqs')[e_1/x_1, \dots, e_l/x_k] = \text{sub}(\text{sub}(eqs')[e_1/x_1])[e_2/x_2, \dots, e_k/x_k]$$

- prove that the substitution preserves the semantics, that is, if $e_j : bt_j$, $j \in [1..k]$, eqs and $\text{var } (x_j : bt_j)_{j \in [1..k]} \text{ in } (x_j = e_j)_{j \in [1..k]}.eqs'$ are semantically equivalent.
- If the substitution succeeds, then the translation preserves the semantics.

Data-structures (tuples, records)

Typing/clocking constraints and normalisation can be extended.

Types and clocks

$$ty ::= ty \times \dots \times ty \mid bt \quad ct ::= ct \times \dots \times ct \mid ck$$

For records, consider that the component must be on the same clock ck .

Normalisation

Rewrites equations into more elementary ones.

E.g.,: rewrite $(x1, x2) = (1, 2)$ into $x1 = 1$ and $x2 = 2$

$(x1, x2) = \text{if } c \text{ then } (1, 2) \text{ else } (3, 4)$ into
 $x1 = \text{if } c \text{ then } 1 \text{ else } 3$ and $x2 = \text{if } c \text{ then } 2 \text{ else } 4$.

The same approach that separates the rewriting from its validation can be followed.

Well Scheduled Equations

Notation $[]$ is an empty sequence of equations; $L = [eq_1; \dots; eq_n]$ is a set of n equations; $eq.L$ is a shortcut for $[eq; eq_1; \dots; eq_n]$.

$Left(ca)$ returns the list of variables that are read in ca .

The predicate $SCH([eq_1; \dots; eq_n]) : r, w, mem$ means:

- the sequence $[eq_1; \dots; eq_n]$ is well scheduled;
- it reads variables in r , write variables in w and memories in mem .

$$\frac{x \notin Left(ca)}{SCH(x = ca) : Left(ca), \{x\}, \emptyset} \quad SCH(x = (v \text{ fby } a)_{bt}^{ck}) : Left(a), \emptyset, \{x\}$$

$$\frac{r = \bigcup_{0 \leq i \leq n} Left(a_i) \cup Left(c) \quad \{y_1, \dots, y_m\} \cap r = \emptyset}{SCH(\vec{y} = f(\vec{a}) \text{ every } c) : r, \{y_1, \dots, y_m\}, \emptyset}$$

$$\frac{SCH(eq) : r, w, mem \quad SCH(L) : r', w', mem' \quad w' \cap r = \emptyset \quad mem \cap r' = \emptyset}{SCH(eq.L) : r \cup r', w \cup w', mem \cup mem'}$$

$$SCH([]) : \emptyset, \emptyset, \emptyset$$

Example (the counting node)

Once the type and clock checking and annotation are done, we get:

```
node counting(tick : bool, top : bool) with (o : int) in (v : int)
  o = (merge top (true → (vb when true(top))ck1)
        (false → (((0 fby ob)b + vb)b when false(top))c)
  and v = (merge tick (true → (1b when true(tick))ck3)
        (false → (0b when false(tick))ck4))b
```

$ck_1 = b$ on true(*top*)

$ck_2 = b$ on false(*top*)

$ck_3 = b$ on true(*tick*)

$ck_4 = b$ on false(*tick*)

We write *b* as a short-cut for base.

Example (the counting node)

After the normalization, it becomes:

```
node counting(tick : bool, top : bool) with (o : int) in (v : int)
  o  =  (merge top (true → (vb when true(top))ck1)
           (false → ((tb + vb)b when false(top))ck2))b
  and t  =  (0 fby ob)b
  and v  =  (merge tick (true → (1b when true(tick))ck3)
           (false → (0b when false(tick))ck4))b
```

where :

$ck_1 = b$ on $\text{true}(top)$

$ck_2 = b$ on $\text{false}(top)$

$ck_3 = b$ on $\text{true}(tick)$

$ck_4 = b$ on $\text{false}(tick)$

Example (the counting node)

After the scheduling, it becomes:

```
node counting(tick : bool, top : bool) with (o : int) in (v : int)
  v = (merge tick (true → (1b when true(tick))ck3)
              (false → (0b when false(tick))ck4))b
  and o = (merge top (true → (vb when true(top))ck1)
                (false → ((tb + vb)b when false(top))ck2))b
  and t = (0 fby ob)b
```

$ck_1 = b$ on true(*top*)

$ck_2 = b$ on false(*top*)

$ck_3 = b$ on true(*tick*)

$ck_4 = b$ on false(*tick*)

Static scheduling and copy variables

The normalisation process may have to introduce extra copy variables; otherwise, equations may not be statically schedulable. E.g.,:

$$\begin{array}{ll} x = 0 \text{ fby } y & \text{and } x = 0 \text{ fby } y \\ \text{and } y = 1 \text{ fby } x & \text{and } y = 1 \text{ fby } cx \end{array}$$

- Both are in normal form but the left sequence is not schedulable.
- Either do a clever normalisation a priori;
- or systematically add a copy for the registers to get an equation of the form: $m = v \text{ fby } x$

Property: if the equation defining x is scheduled after any read of m , then x and m can be stored in the same location.

By characterising well scheduled equations only, we give the liberty to the compiler to consider several possible implementations of the scheduling/normalization functions.

Translation to Sequential Code

We introduce an intermediate target imperative language in which annotated normalized data-flow programs are compiled.

What do we need?

- represent transition functions in an imperative style
- a simple memory model: static allocation of memory; no aliasing.
- such that the translation into C code is simple.

Intuition

A synchronous function f defines a “class” with

- a set of state variables and a set of instance variables;
- a set of methods that read/write these state variables.

The memory model is a tree and there is no aliasing between states. The method of a class can only modify its own states (“instance variables”).

A Simplification

For MiniLS, we only need to produce a class with two methods `step` and `reset`:

- Given the current inputs, the method `step` produces the current outputs and modifies in place its internal state.
- A method `reset` initialize/reset its internal state.

Yet, the general case with several methods is useful. E.g.,:

- provides `set/get` methods to have direct access to inputs and output in order to reduce the number of copies;
- to implement the modular static scheduling problem;
- to do program specialisation, e.g., implement a special faster method for particular values of the inputs.
- to implement the language extended with ODEs (see related course).

The Obc Intermediate Language

md ::= $\text{let } x = c \mid md; md$
 $\text{let } f = \text{class} \langle M, I, (\text{method}_i(p_i) = q_i \text{ where } S_i)_{i \in [1..n]} \rangle$

p, q ::= $x : bt, \dots, x : bt$

M ::= $[x : bt [= v]; \dots; x : bt [= v]]$

I ::= $[o : f; \dots; o : f]$

c ::= $v \mid lv \mid \mid op(c, \dots, c) \mid o.method(c, \dots, c)$

S ::= $() \mid lv \leftarrow e \mid S ; S \mid \text{var } x : bt \text{ in } S \mid \text{if } c \text{ then } S \text{ else } S$
 $\mid \text{case } (x) \{ C : S; \dots; C : S \}$

R, L ::= $S; \dots; S$

lv ::= $x \mid \text{state}(x)$

$method$::= $step \mid reset \mid \dots$

Principles of the translation

- Hierarchical memory model which corresponds to the call graph: one instance variable per function call;
- Control-structure (invariant): an equation annotated with clock ck is executed when ck is true.
- A guarded equations $x = e^{ck}$ translates into a control-structure. E.g., the equation:

$$x = (y + 2)^{\text{base on } C_1(x_1) \text{ on } C_2(x_2)}$$

is translated into a piece of control-structure:

$$\text{case } (x_1) \{ C_1 : \text{case } (x_2) \{ C_2 : x = y + 2 \} \}$$

- The translation is made by a linear traversal of the sequence of normalized/scheduled equations.

- local generation of a control-structure from a clock:

$$\begin{aligned} \text{Control}(\text{base}, S) &= S \\ \text{Control}(ck \text{ on } C(x), S) &= \text{Control}(ck, \text{case } (x) \{ C : S \}) \end{aligned}$$

- merge them locally

$$\begin{aligned} &\text{Join}(\text{case } (x) \{ C_1 : S_1; \dots; C_n : S_n \}, \text{case } (x) \{ C_1 : S'_1; \dots; C_n : S'_n \}) \\ &= \text{case } (x) \{ C_1 : \text{Join}(S_1, S'_1); \dots; C_n : \text{Join}(S_n, S'_n) \} \\ &\text{Join}(S_1, S_2) = S_1; S_2 \end{aligned}$$

Control-optimization:

- The **scheduling** function must put equations with the same clock or one that is a sub-clock of the other close to each others
- $ck \text{ on } C(x)$ is a subclock of ck' if ck is a subclock of ck' or $ck = ck'$.

Example

```
class counting =  
  memory t1 : int = 0;  
  
  reset () = state(t1) := 0;  
  
  step(tick:bool,top:bool) returns (o:int) =  
    v:int, t2:int in  
    case (tick) {  
      | True: v := 1;  
      | False: v := 0; };  
    case(top) {  
      | True: o := v;  
      | False: o := state(t1) + v; };  
    t2 := o;  
    state(t1) := t2;
```


Example (modularity)

- A node definition is compiled separately, once for all.
- A node that calls an other node need a memory to store its internal state.

Example:

```
node sum(x:int) returns (o:int) with  
  o = 0 fby o + x;
```

```
node conduct(c:bool;input:int) returns (o:int) with  
var o':int in  
  o = merge c (true -> o')  
            (false -> (0 fby o) when false(c)) and  
  o' = sum(input when true(c))
```

Target code:

```
class conduct =  
  memory x_2 : int = 0  
  instances x_4 : sum  
  
  reset() =  
    x_4.reset();  
    state x_2 := 0;  
  
  step(c : bool; input : int) returns (o : int)  
    var o' : int in  
    case (c) {  
      case true :  
        o' := x_4.step(input);  
        o := o';  
      case false :  
        o := state(x_2);  
    };  
    state x_2 := o; }
```

Demo

Velus, Heptagon⁵ and Scade 6

⁵Available at <http://heptagon.gforge.inria.fr>, with source code in OCaml.

Notations

- If $p = [x_1 : bt_1; \dots; x_n : bt_n]$ and $p_2 = [x'_1 : bt'_1; \dots; x'_k : bt'_k]$ then $p_1 + p_2 = [x_1 : bt_1; \dots; x_n : bt_n; x'_1 : bt'_1; \dots; x'_k : bt'_k]$ provided $x_i \neq x'_j$ for all i, j such that $1 \leq i \leq n, 1 \leq j \leq k$.
- $[]$ denotes the empty list of variable declarations.
- m_1 and m_2 denotes environments for memories.
- j_1 and j_2 denotes environments for instances.
- $m_1 + m_2$ for the composition of two substitutions on memory names and $j_1 + j_2$ on object instances.
- $S \cdot L$ is a list of instructions whose head is S and tail is L . $[]$ is the empty list and $[S_1; \dots; S_n] = S_1 \cdot (\dots \cdot S_n \cdot [])$.

Translation functions

Applied on normalised/scheduled expressions and equations.

- $TE_m(a)$ translates an expression a .
- $TCA_m(y, ca)$ translates an expression ca with result stored in y .
- $TEq_{\langle m, S, j, L \rangle}(eq)$ defines the translation of an equation:
 - m is a memory environment;
 - S is executed when reset;
 - j is the instance environment;
 - L is a sequence of instructions
- $TEList_m[a_1; \dots; a_n]$ translates a list of expressions.
- $TEqList([eq_1; \dots; eq_n])$ translates a list of equations.

Expressions

$$TE_m(e_{bt}^{ck}) = TE_m(e)$$

$$TE_m(v) = v$$

$$TE_{m+[x:bt=v]}(x) = \text{state}(x)$$

$$TE_m(x) = x \text{ otherwise}$$

$$TE_m(a \text{ when } C(x)) = TE_m(a)$$

$$TE_m(op(a_1, \dots, a_n)) = \text{let } [c_1; \dots; c_n] = TEList_m[a_1; \dots; a_n] \text{ in } op(c_1, \dots, c_n)$$

Controlled expressions

$$TCA_m(y, \text{merge } x (C \xrightarrow{\rightarrow} ca)_{bt}^{ck}) = \text{case } (x) \{ C : TCA_m^{\rightarrow}(y, ca) \}$$

$$TCA_m(y, a) = y := TE_m(a) \text{ otherwise}$$

Equations

$$\begin{aligned}TEList_m[a_1; \dots; a_n] &= [TE_m(a_1); \dots; TE_m(a_n)] \\TEqList(eq) &= TEq_{\langle [], \text{skip}, [], [] \rangle}(eq) \\TEqList([eq_1; \dots; eq_n]) &= TEq_{TEqList([eq_2; \dots; eq_n])}(eq_1)\end{aligned}$$

Instantaneous equations, synchronous register

$$TEq_{\langle m, S, j, L \rangle}(x = e_{bt}^{ck}) = \langle m, j, S, (\text{Control}(ck, TCA_m(x, e_{bt}^{ck}))) \cdot L \rangle$$

$$\begin{aligned}TEq_{\langle m, S, j, L \rangle}(y = (v \text{ fby } a)_{bt}^{ck}) = \\ \text{let } m' = m + [y : bt = v] \text{ in} \\ \text{let } c = TE_{m'}(a) \text{ in} \\ \langle m', \text{state}(y) := v; S, j, (\text{Control}(ck, \text{state}(y) := c)) \cdot L \rangle\end{aligned}$$

Application of a node

$$\begin{aligned} TEq_{\langle m, S, j, L \rangle} ((x_1, \dots, x_k) = f(a_1, \dots, a_n) \text{ every } e_{bt}^{ck}) = \\ \text{let } c_0 = TE_m(e_{bt}^{ck}) \text{ in} \\ \text{let } [c_1, \dots, c_n] = TEList_m[a_1; \dots; a_n] \text{ in} \\ \langle m, o.\text{reset}; S, [o : f] + j \\ (\text{Control}(ck, \text{case } (c_0) \{(\text{true} : o.\text{reset})\})) \cdot \\ (\text{Control}(ck, (x_1, \dots, x_k) = o.\text{step}(c_1, \dots, c_n))) \cdot L \rangle \text{ where } o \notin Dom(j) \end{aligned}$$

Translation of a node definition

```
TP (node  $f(p)$  returns ( $q$ ) var  $r$  in  $eq_1$  and ...  $eq_n$ ) =  
  let  $\langle m, S, j, L \rangle = TEqList([eq_1; \dots; eq_n])$  in  
  class  $f = \langle$ memory =  $m$ ;  
             instances =  $j$ ;  
             reset =  $S$ ;  
             step( $p$ ) returns( $q$ ) var  $r$  in  $JoinList(L)$  $\rangle$   
  where  $SCH([eq_1; \dots; eq_n])$ 
```

We write $SCH([eq_1; \dots; eq_n])$ when there exists r, w, m such that
 $SCH([eq_1; \dots; eq_n]) : r, w, m$.

From Obc to a target language

The translation from Obc to C is straightforward. Yet, the proof of correctness is not [3].

Principle

- For a class f , define a C structure that stores the internal state of f .
- Every method m of f becomes a function f_m with an extra input *self* that point to its internal state.
- When the “step” method has several outputs, returns a C structure on the stack or, better:
- define a C structure to store the output; the method takes an extra pointer to it.
- or add extra methods, with no output, to get the value of each output. They are implemented in C by direct access to the state of the callee.

Demo

Velus, Heptagon and Scade 6

Optimizations

- Some optimizations can be done by the compiler of the target language. E.g., it is useless to optimize reuse between local (stack) variables.
- But this depends on the quality of the compiler of the target language.
- Some classical optimizations (CSE, copy and const. prop., inlining) can be applied directly on the data-flow representation.
- It is always useful to reduce the number of state variables and the liveness between reads and writes.
- Optimize the control structure. E.g., gather if/then/else.
- These two optimizations depend on the scheduling heuristic.

Optimization that a C compiler cannot do easily

- Avoid copies: x and $\text{pre } x$ can be shared when all equations reading $\text{pre } x$ can be scheduled before the equation $x = \dots$. E.g.:
 $x = \text{pre } x + 1$ can be compiled into $x := x + 1$
- Automata minimization (generalization of CSE). E.g.,
 $y = 0 \text{ fby } y + 1$ and $z = 0 \text{ fby } z + 1$ as
 $y = 0 \text{ fby } y + 1$ and $z = y$.
- share memories between two pieces of code never active in parallel and when going from one to the other is reset on entry.
- for array iterators, perform loop fusion and generate in place modification for functional updates. [6]

Those optimisations are implemented in the Heptagon compiler.

Control Optimization

Share/reduce the number of control-structures

- some piece of code is only executed at the very first instant or only when some condition is true. E.g.,

```
if state(i) then x = 0; /* initialization code */
...
if state(i) { y = 1;} /* initialization code */
...
if c then x = m -> pre_1 + 1;
    /* step when clock c_1 is true */
...
if c then m -> pre_1 = x;
    /* set the memory when clock c_1 is true */
```

- minimize the number of if/then/else to open. For that, define a scheduling function which gather equations activated on the same clock (cf. *Join(.,.)*).

Still, the generation is not that efficient.

Compilation into Automata

Generating a single step functions means that some conditions that are surely false will be executed at every step. E.g., consider the way an initialization $o = x \rightarrow y$ is compiled. In Obc.

```
if state(init_1) then o := x else o := y;  
...;  
state(init_1) := false;
```

Generates an “optimal” control structure which only execute the necessary code at every instant.

This is the idea of **compilation into automata** introduced by Halbwachs and Plaice (Lustre V2).

It was improved to generate a minimal automaton (Lustre V3) by Ratel et Raymond in 1991 [7].

It is implemented in the academic Lustre compiler.

An example (in Lustre syntax)

```
node counter(tick,top:bool) returns (cpt:int)
  var i:int;
  let cpt = 0 -> if pre top then i
                  else if tick then pre cpt + 1
                  else pre cpt;
    i = if tick then 1 else 0;
  tel;
```

After normalization and scheduling, we get:

```
node counter(tick,top:bool) returns (cpt:int)
  var i:int;
  let i = if tick then 1 else 0;
      cpt = if init then 0
              else if ptop then i
              else if tick then pcpt + 1
              else ptop;
      ptop = pre top;
      pcpt = pre cpt;
      init = true fby false
  tel;
```


Single loop code

```
if tick then i := 1 else i := 0;  
if state(init) = 0 then cpt := 0  
else if state(ptop) then cpt := i  
else if tick then cpt := state(pcpt) + 1  
      else cpt := state(pcpt)  
state(init) := false;  
state(ptop) := top;  
state(pcpt) := cpt
```

top and state(ptop) can be stored at the same location; the same with cpt and state(pcpt). Then, the last two assignment can be removed (see remark on slide 34).

Example

Initial state: $S_1 = [true/init]$

The code that computes the output is:

```
if tick then i := 1 else i := 0;  
cpt := 0;  
state(pcpt) := cpt
```

- It can be simplified into `state(pcpt) := 0;`
- The code that computes the next state is:

```
if top then state(ns) := S2 else state(ns) := S3;
```

State S_2 : $S_2 = [false/init, true/ptop]$

```
if tick then i := 1 else i := 0;  
state(pcpt) := i;  
if top then state(ns) := S2; else state(ns) := S3;
```

State S_3 : $S_3 = [false/init, false/ptop]$

```
if tick then  
  { i := 1;  
    state(pcpt) := state(pcpt) + 1; }  
else  
  i := 0;  
if top then state(ns) := S2  
else state(ns) := S3;
```

The final automaton

```
case state(ns){  
  S1: state(pcpt) := 0;  
      if top state(ns) := S2; else state(ns) := S3;  
  S2: if tick then i := 1 else i := 0;  
      state(pcpt) := i;  
      if top then state(ns) := S2 else state(ns) := S3;  
  S3: if tick then state(pcpt) := state(pcpt) + 1;  
      if top state(ns) := S2 else state(ns) := S3;  
}
```

Conclusion

- far better code but the size has increased
- assertions (i.e., `assert P` in **Lustre**) can be taken into account during the enumeration

Problems

- combinatorial explosion
- in **Lustre**, the control-structure is hidden and encoded with booleans (think of a one-hot encoding of an automaton)
- which boolean variables should we consider? There is no *good* programming rules to avoid this explosion
- limit to a predefined set of boolean variables (e.g., clocks).

Solutions?

- automata minimization done a posteriori.
- direct generation of a minimal automaton (called “compilation on demand”, [Halbwachs, Ratel, Raymond, PLILP 91])

An example (Halbwachs et al [7])

```
node Example(i: bool) return (n: int);  
var x,y,z : bool;  
let  
  n = 0 -> if (pre x) then 0 else (pre n) + 1;  
  x = false -> not (pre x) and z;  
  y = i -> if (pre x) then (pre y) and i  
           else (pre z) or i;  
  z = true -> if (pre x) then (pre z)  
              else ((pre y) and (pre z)) or i);  
tel
```

4 state variables (->, pre x, pre y and pre z)

Example

Example:

- $q_0 : (init, pre_x, pre_y, pre_z) = (1, nil, nil, nil)$

Action: $n:=0$

$next_{init}(q_0, i) = 0$

$next_{pre_x}(q_0, i) = 0$

$next_{pre_y}(q_0, i) = i$

$next_{pre_z}(q_0, i) = 0$

if (i) { state = q_1 ; } else { state = q_2 ; }

with $q_1 = (0, 0, 1, 1)$ et $q_2 = (0, 0, 0, 1)$

- $q_1 = (0, 0, 1, 1)$

Action: $n:=n+1$

$next_{init}(q_1, i) = 0$

$next_{pre_x}(q_1, i) = 1$

$next_{pre_y}(q_1, i) = 1$

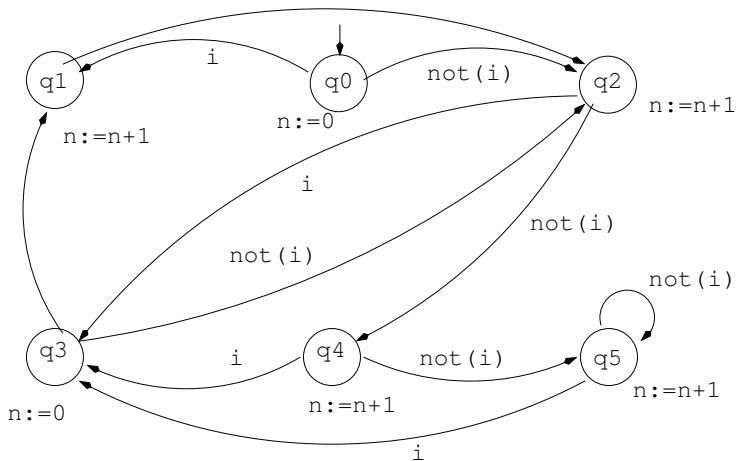
$next_{pre_z}(q_1, i) = 1$

state = q_3

- $q_2 = (0, 0, 0, 1)$

Action: $n:=n+1$, if (i) { state = q_3 ; } else { state = q_4 ; }

- $q_3 = (0, 1, 1, 1)$
Action: $n:=0$, if (i) {state = q_1 ; } else { state = q_2 ; }
- $q_4 = (0, 0, 1, 0)$
Action: $n:=n+1$, if (i) { state = q_3 ; } else { state = q_5 ; }
- $q_5 = (0, 0, 0, 0)$
Action: $n:=n+1$, if (i) { state = q_3 ; } else { state = q_5 ; }



Compilation into automata “on demand”

- the automaton is not minimal: q_0 and q_3 are equivalent; q_4 , q_5 and q_2 are equivalent
- we can minimize *a posteriori* (**Lustre V2**) but still an explosion of the number of states in the intermediate automaton

Solution: directly generate a minimal automaton. In practice, the code is still too big, but:

- this technique say something very interesting when the main node has a single boolean variable
- what is the minimal automaton for a node with a single boolean variable which is always true? The trivial automaton true!
- this corresponds exactly to proving a safety (invariant) property by a Model Checking technique.

Conclusion

- The compilation into automata is possible but it is not modular and tend to generate enormous code.
- The compiler of **Scade** generate single-loop code.
- The clock-directed translation method is a good compromise (simple et reasonably efficient code).
- If the input language has automata, like **Scade 6**, the result of compilation into automata could be expressed as a source-to-source transformation.
- Would-it simplify the proof of its correctness?

References I



Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet.

Clock-directed Modular Code Generation of Synchronous Data-flow Languages.

In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.



Sylvain Boulmé and Grégoire Hamon.

Certifying Synchrony for Free.

In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag.

Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at www.di.ens.fr/~pouzet/bib/bib.html.



Timothy Bourke, Lélío Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg.

A Formally Verified Compiler for Lustre.

In *International Conference on Programming Language, Design and Implementation (PLDI)*, Barcelona, Spain, June 19-21 2017. ACM.



Paul Caspi and Marc Pouzet.

Synchronous Kahn Networks.

In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pennsylvania, May 1996.



Jean-Louis Colaço and Marc Pouzet.

Clocks as First Class Abstract Types.

In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.



Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet.

A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler.

In *Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, Beijing, June 12-13 2012. ACM.

Best paper award.

References II



N. Halbwachs, P. Raymond, and C. Ratel.

Generating efficient code from data-flow programs.

In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.



R. Lublinerman, C. Szegedy, and S. Tripakis.

Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size.

In *ACM Principles of Programming Languages (POPL)*, 2009.



Marc Pouzet and Pascal Raymond.

Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation.

In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.



Pascal Raymond.

Compilation séparée de programmes Lustre.

Technical report, Projet SPECTRE, IMAG, July 1988.



Pascal Raymond.

Compilation efficace d'un langage déclaratif synchrone: le générateur de code Lustre-v3.

PhD thesis, Institut National Polytechnique de Grenoble, 1991.