Synchronous languages

Lecture 4:
Formal aspects of the LUSTRE language

ENSEEIHT 3A – parcours E&L
2021/2022

Frédéric Boniol (ONERA)
frederic.boniol@onera.fr

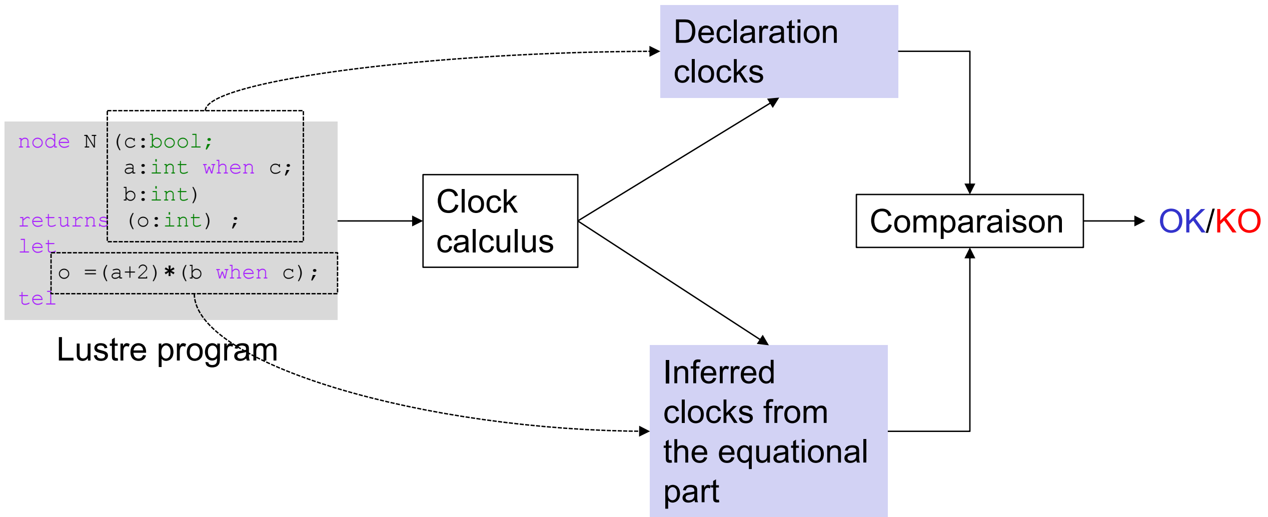
LUSTRE formal aspects

- LUSTRE is simple enough to be defined by a formal semantics
  – The behaviour of a LUSTRE program is formally defined by a set of mathematical functions

- Advantage
  – Certification of the code generator (LUSTRE to C and Ada)
  – Capacity to
    - Formally check the consistency of a LUSTRE program
    - Formally check safety properties

⇒ Plan of lecture 4
  – 4.1. Clock calculus (to check that a program is well synchronized)
  – 4.2. Formal verification of safety properties

4.1. Clock calculus



LUSTRE Syntax (recall)

- Declarative part

```
X : type;
X : type when B ;
const X : type = val;
```

For a given program P, let us call *input*, *local*, *output,* and *constant* the sets of inputs flow declarations, output flow declarations, local flow declarations and constant flow declarations.

- Equational part

```
Y = exp ;
```

```
exp ::= f(exp₁,…, expₙ)          (with f = +, -, *, /, or…)
       | exp₁ when exp₂
       | current(exp)
       | pre(exp)
       | exp₁ -> exp₂
       | X                       (flow)
       | val                     (literal value)
```

For a given program P, let us call *Eq* and *Lit* the sets of equations and literal values of P

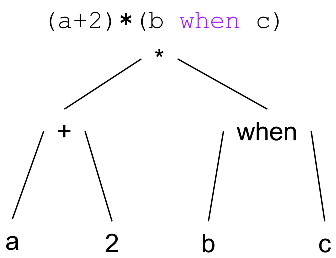
Syntax: example

- Example

  Let us consider the following program

```
node N (c:bool; a:int when c; b:int)
returns (o:int) ;
let
   o =(a+2)*(b when c);
tel
```

- inputs = {"c:bool", "a:int when c", "b:int"}
- outputs = {"o:int"}
- locals = ∅
- constants = ∅
- Eq = {"o =(a+2)*(b when c)"}
- Lit = {"2"}



Abstract syntax tree of the program

Clock functions

Notations:

- clk_dec : flow -> Boolean expression ∪ {all}

  clk_dec is a fonction which associates each flow with its declaration clock.

- clk : exp -> Boolean expression ∪ {all}

  clk is a function which associates each flow with its clock inferred from the equational part of the program.

=> Definition of clk_dec and clk by a set of inference rules

Clock calculus: inference rules

1. $$\frac{\text{"X : type"} \in \ \textit{input} \cup \textit{output} \cup \textit{local}}{\text{clk\_dec(X) = true}}$$

2. $$\frac{\text{"X : type } \texttt{when} \text{ B"} \in \ \textit{input} \cup \textit{output} \cup \textit{local}}{\text{clk\_dec(X) = B}}$$

3. $$\frac{\text{"}\texttt{const}\text{ X : type = val"} \in \ \textit{constant}}{\text{clk\_dec(X) = } \texttt{all}}$$

4. $$\frac{\text{"X : type"} \in \ \textit{input} \ \vee \ \text{"X : type } \texttt{when} \text{ B"} \in \ \textit{input}}{\text{clk(X) = clk\_dec(X)}}$$

5. $$\frac{\text{val} \in \text{Lit}}{\text{clk(val) = } \texttt{all}}$$

Clock calculus: example

- Example

```
node N (c:bool; a:int when c; b:int)
returns (o:int) ;
let
   o =(a+2)*(b when c);
tel
```

- inputs = {"c:bool", "a:int when c", "b:int"}
- outputs = {"o:int"}
- locals = Ø
- constants = Ø
- Lit = {"2"}

Rules 1 and 2 →

clk_dec(c) = true
clk_dec(a) = c
clk_dec(b) = true
clk_dec(o) = true

Rules 4 and 5 →

clk(c) = true
clk(a) = c
clk(b) = true
clk(2) = all

Clock calculus: inference rules

6. $$\frac{\text{clk}(\text{exp}_1) = \texttt{all} \ \wedge \ldots \wedge \text{clk}(\text{exp}_n) = \texttt{all}}{\text{clk}(\text{f}(\text{exp}_1,\ldots, \text{exp}_n)) = \texttt{all}}$$

7. $$\frac{\text{clk}(\text{exp}_1) \in \{\text{b}, \texttt{all}\} \wedge \ldots \wedge \text{clk}(\text{exp}_n) \in \{\text{b}, \texttt{all}\} \wedge \exists \text{i clk}(\text{exp}_\text{i}) \neq \texttt{all}}{\text{clk}(\text{f}(\text{exp}_1,\ldots, \text{exp}_n)) = \text{b}}$$

8. $$\frac{}{\text{clk}(\texttt{pre}(\text{exp})) = \text{clk}(\text{exp})}$$

9. $$\frac{\text{clk}(\text{exp}_1) = \texttt{all} \wedge \text{clk}(\text{exp}_2) = \texttt{all}}{\text{clk}(\text{exp}_1 \text{ -> } \text{exp}_2)) = \texttt{all}}$$

10. $$\frac{\text{clk}(\text{exp}_1) \in \{\text{b}, \texttt{all}\} \wedge \text{clk}(\text{exp}_2) \in \{\text{b}, \texttt{all}\} \wedge \exists \text{i clk}(\text{exp}_\text{i}) \neq \texttt{all}}{\text{clk}(\text{exp}_1 \text{ -> } \text{exp}_2)) = \text{b}}$$

Clock calculus: inference rules

11. $$\frac{\text{clk}(\text{exp}_1) \in \{\text{b}, \texttt{all}\} \ \text{ and clk}(\text{exp}_2) \in \{\text{b}, \texttt{all}\}}{\text{clk}(\text{exp}_1 \texttt{ when } \ \text{exp}_2)) = \text{exp}_2}$$

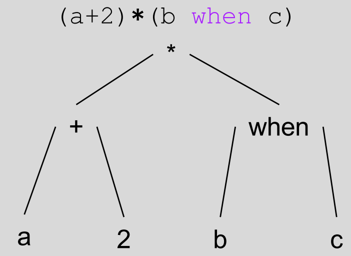
12. $$\frac{\text{clk}(\text{exp}) \neq \texttt{all} \ \text{ and } \ \text{clk}(\text{exp}) \neq \text{true}}{\text{clk}(\texttt{current}(\text{exp})) = \text{clk}(\text{clk}(\text{exp}))}$$

13. $$\frac{\text{"Y = exp"} \in \text{Eq}}{\text{clk}(\text{Y}) = \text{clk}(\text{exp})}$$

Clock calculus: example

- Example

```
node N (c:bool; a:int when c; b:int)
returns (o:int) ;
let
   o =(a+2)*(b when c);
tel
```

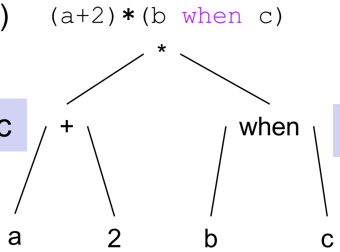


Abstract syntax tree of the program

clk_dec(c) = true
clk_dec(a) = c
clk_dec(b) = true
clk_dec(o) = true

clk(c) = true
clk(a) = c
clk(b) = true
clk(2) = all

clk((a+2)*(b when c)) = c
(by rule 7)



clk(a+2) = c
(by rule 7)

clk(b when c) = c
(by rule 11)

Clock calculus: example

- Example

```
node N (c:bool; a:int when c; b:int)
returns (o:int) ;
let
   o =(a+2)*(b when c);
tel
```

clk_dec(c) = true
clk_dec(a) = c
clk_dec(b) = true
clk_dec(o) = true

clk(c) = true
clk(a) = c
clk(b) = true
clk(2) = all
clk(a+2) = c
clk(b when c) = c
clk((a+2)*(b when c)) = c

clk(o) = c
(by rule 13)

Eq = {"o =(a+2)*(b when c)"}

Clock calculus: correctness

Definition: well synchronized program

Let P be a LUSTRE program. P is well synchronized iff

- clk_dec(X) has been inferred for all X ∈ *flows(P)*
- clk(X) has been inferred for all X ∈ *flows(P)*
- clk(X) = clk_dec(X) for all X ∈ *flows(P)*

Clock calculus: example

Exercise:

Let us consider the following program.

```
node N (c:bool; a:int when c; b:int)
returns (o:int) ;
let
   o =(a+2)*(b when c);
tel
```

Question: Is this program well synchronized? Otherwise fix it.

Answer: No. Because clk_dec(o) ≠ clk(o)

=> Two possible corrections

First correction

```
node N (c:bool;
        a:int when c;
        b:int)
returns (o:int int when c) ;
let
   o =(a+2)*(b when c);
tel
```

First correction

```
node N (c:bool;
        a:int when c;
        b:int)
returns (o:int) ;
let
   o = current (a+2)*(b when c);
tel
```

Clock calculus: exercice

Exercise: complete the following program with the correct clock declarations!

```
node prog1 (B1, B2 : bool;
               H1 : bool when B1;
               H2 : bool when B2;
               X1 : int when H1;
               X2 : int when H2;
               Y : int when (B1 and B2))
returns (S : int when ...)
var Z1 : int when …; Z2 : int when ...;
     Z3 : int when ...;Z4 : int when ...;
let Z1 = current(X1);
     Z2 = current(X2);
     Z3 = current(Z1) + current(Z2);
     Z4 =(Z3 when (B1 and B2)) + Y;
     S = current(Z4);
tel
```


4.2. Formal verification of LUSTRE programs

Verification of LUSTRE programs: idea (1/3)

Idea 1

- LUSTRE allow to express sequences of values (data flows)

⇒ LUSTRE can be seen as past temporal logic

⇒ Idea:

  - Express the properties to check in LUSTRE
  - Check them using the LUSTRE code generation

Principle

- Write the property to check as a LUSTRE node
- Compose this node with the program to verify
- Look at the possible states reached by this new system

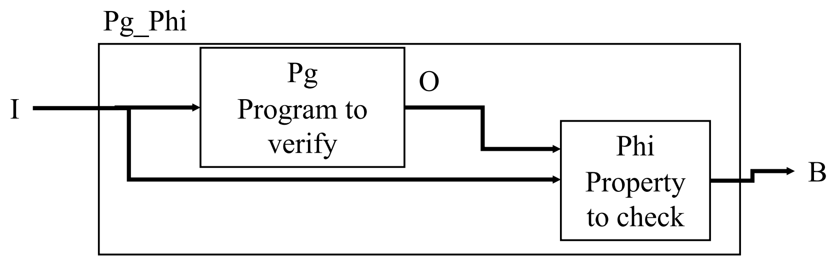
Restriction

LUSTRE can only specify safety properties (i.e., properties that are always true, or never true)

Verification of LUSTRE programs: idea (2/3)

- Pg = program to verify (in LUSTRE)
- Phi = property to check, written as an observer node in LUSTRE

=> Pg_Phi = Pg || Phi   = new LUSTRE program



⇒ Phi is satisfied iff ssi all the reachable states from the intial state Pg_Phi are labelled by val(B)=true

Verification principle:

Generation of all the reachable states of Pg_Phi

Stop as soon as val(B)=false or all the reachable states have been visited.

Tools

LESAR (LUSTRE tool suit), NP-tools (SCADE tool suit)

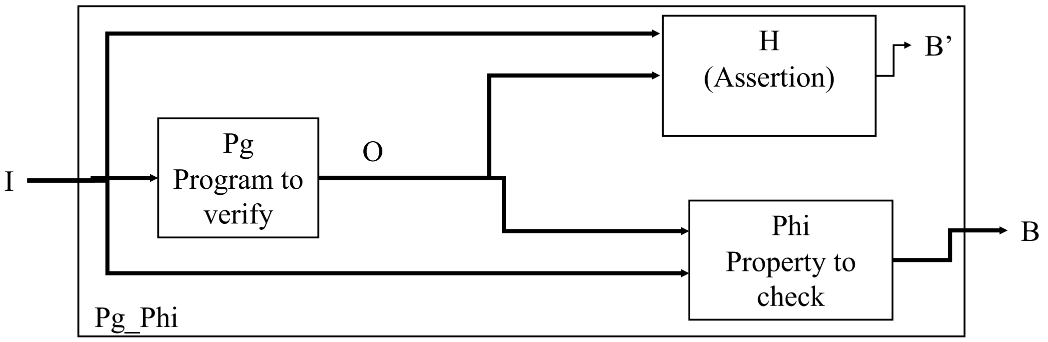
Verification of LUSTRE programs: idea (3/3)

Idea 2

– Use of assertions in order to reduce the space of reachable states

⇒ Allow to model hypothesis on the environment of the system

– Recall: assertion is a Boolean expression that is supposed to be always true



– H = assertion on the behavior of the system and the enviroment

– The states to explore are those satisfying val(B')=true

Notion of observator

Obervator:

A node which encodes a property to check

⇒ Contains only one output flow

⇒ This output flow denotes the Boolean value of the property

Example :

```
node never (A : bool) returns (B : bool);
let
  B = not A -> not A and pre(B);
tel.
```

Encode the property Never(X) =

for all n in the sampling clock of X, X_n is false

As soon as X becomes true, Never(X) becomes false.

Notion of observator

Example :

Let us consider the following property:

"Every occurrence of A must be followed by an occurrence of B before the next occurrence of C"

=> Expressed in a past style:

"At each occurrence of C, either A has never occurred, or if A has occurred, B must have occurred since the last occurrence of A"

```
node once_B_from_A_to_C (A,B,C:bool)returns (X:bool);
let
  X = C => (never(A) or since(B,A));
tel.

node since (X,Y:bool) returns (Z:bool);
let
  Z = if Y then X
      else (true -> X or pre(Z));
tel.
```


LUSTRE tool suit

- URL of the LUSTRE tool suit

  http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/

The distribution is available for the following platforms:

Linux 64 (x86_64)
Linux 32 (ix86)
MacOSX (x86_64)
Cygwin/X 64(x86_64)
Cygwin/X 32(ix86)

Tools to use for the practical session

– luciole : simulation of LUSTRE nodes
– lesar : model checker for LUSTRE nodes

End of lecture 4