Synchronous Languages 101
A Story

Laure Gonnord and Lionel Morel
University of Lyon/ LIP - INSA Lyon / CITI
March 2016

1By 3rd generation degenerates
Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The Lustre language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?
Safety-Critical Embedded Systems
Reactive System: Example

Pitch
Yaw
Roll

pilot command
aircraft behavior
order for
alarms
Governs controlled
every 1 ms

Controller
Mission: Stabilize a
naturally unstable aircraft

...I_i...I_2, I_1, I_0

...O_i...O_2, O_1, O_0
Controller
Mission: Stabilize a naturally unstable aircraft

pilot command
aircraft behavior
order for governs
alarms

...$I_i ...I_2, I_1, I_0$
...$O_i ...O_2, O_1, O_0$

1 ms!

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 $M_{i+1} = g(I_i, M_i)$
Difficulties

- Inputs do not arrive simultaneously, so:
  - When do we start computing?
  - How to interpret a late input: Absent? Delayed?
- The controller usually comprises several activities
  ⇒ how to schedule them?
- WCET estimation is hard!
The Asynchronous Approach

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

```c
int T1(){ // prio = P1
    while(true){
        readSensor();
        setActuator1();
        wait(100ms);
    }
}

int T2(){ // prio = P2
    while(true){
        lock(1);
        readSensor();
        updateState();
        unlock(1);
        setActuator2();
    }
}

int T3(){ // prio = P3
    while(true){
        whenever(event){
            lock(1);
            readState();
            updateState();
            unlock(1)
            setActuator2();
        }
    }
}

void scheduler(){
    if(Round_Robin){
        ...
    }
    if(FIFO){
        ...
    }
    if(PRIO){
        nextT = f(P1,P2,P3);
    }
}
```

- Problem #1: **Inter-task communications**
- Problem #2: **Scheduling**

$\Rightarrow$ **Globally non-deterministic!!**
Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The Lustre language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?
Requirements

- Design correctness: because safety is crucial!
- Need to reason formally about the operations of the system
  ⇒ Design methodologies and tools built on solid mathematical foundations
Let’s Design Languages That ... 

- Natively support **deterministic functional concurrency**
- **Programming style** natural for engineers
- Build on a **simple formal model**
- Can be compiled down to executable code
- Follow **synchronous execution** philosophy

```plaintext
Initialize Memory
for each input event do
  Compute Outputs
  Update Memory
end
```

```plaintext
Initialize Memory
for each clock tick do
  Read Inputs
  Compute Outputs
  Update Memory
end
```
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:
  - ⇒ We can ignore time inside an instant, only the order matters;
  - ⇒ We are only interested in how instants are chained together.

Programming Styles Natural for Engineers
Essentially: block diagrams and FSMs.
Synchronous Languages vs Others

Advantages

• **Semantics defined formally** ⇒ enables formal proofs and provable compilation;

• **High abstraction** level ⇒ less work for the programmer, more for the compiler;

• **Bounded** memory and execution time;

• Barely needs an OS: **scheduling is defined in the language’s semantics**

Disadvantages

• Produced code less efficient than hand-written code;

• Synchronous hypothesis hard to ensure (WCET not solved, distributed systems);

• Not well-suited for multi-rate systems.
Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The Lustre language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?
Flows and Clocks

- **Lustre** is a **data-flow** language: every expression and variable is a flow;
- **Flow**: infinite sequence of values + clock;
- **Clock**: defines when a flow is present (has a value).

**Example**

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>...</td>
</tr>
<tr>
<td>y</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nb: for today, we forget about clocks
Point-Wise Extension of Classic Operators

Example

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>True/False</td>
<td>True/False</td>
<td>True/False</td>
<td>False/False</td>
<td>...</td>
</tr>
<tr>
<td>x</td>
<td>x_0</td>
<td>x_1</td>
<td>x_2</td>
<td>x_3</td>
<td>...</td>
</tr>
<tr>
<td>y</td>
<td>y_0</td>
<td>y_1</td>
<td>y_2</td>
<td>y_3</td>
<td>...</td>
</tr>
<tr>
<td>x+y</td>
<td>x_0 + y_0</td>
<td>x_1 + y_1</td>
<td>x_2 + y_2</td>
<td>x_3 + y_3</td>
<td>...</td>
</tr>
<tr>
<td>if c then x else y</td>
<td>x_0</td>
<td>y_1</td>
<td>x_2</td>
<td>y_3</td>
<td>...</td>
</tr>
</tbody>
</table>
Delay Operator

• The \textit{pre}(x) operator denotes the \textit{previous value} of a flow \( x \);
• Undefined for the first instant;
• Usually combined with the initialization operator \(\rightarrow\): \( x \rightarrow y \) is \( x \) on the first instant, \( y \) otherwise.

Example

\begin{center}
\begin{tabular}{l|l|l|l|l|l|l}
\hline
\textbf{time} & 0 & 1 & 2 & 3 & \ldots \\
\hline
\textbf{x} & \( x_0 \) & \( x_1 \) & \( x_2 \) & \( x_3 \) & \ldots \\
\hline
\textbf{y} & \( y_0 \) & \( y_1 \) & \( y_2 \) & \( y_3 \) & \ldots \\
\hline
\textbf{pre \ x} & \( \text{\textit{pre}} \ x_0 \) & \( x_1 \) & \( x_2 \) & \ldots \\
\hline
\textbf{\( y \rightarrow \text{\textit{pre}} \ x \)} & \( y_0 \) & \( \text{\textit{pre}} \ x_0 \) & \( x_1 \) & \( x_2 \) & \ldots \\
\hline
\end{tabular}
\end{center}
Example: A Resettable Counter

\[ \text{node} \ \text{counter}(\text{reset}: \text{bool}) \ \text{returns} \ (\text{count}: \text{int}) \]

\[
\text{let} \\
\quad \text{count} = 0 \rightarrow \text{if} \ \text{reset} \ \text{then} \ 0 \ \text{else} \ \text{pre}(\text{count})+1; \\
\text{tel}
\]
Example: A Resettable Counter

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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example: A Resettable Counter

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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example: A Resettable Counter

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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>False</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example: A Resettable Counter

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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: A Resettable Counter

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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Example: A Resettable Counter

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

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>...</td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>
Running Example

Let’s:

- count modulo 9
- and display counter on a 7-seg
- reset if a button is pressed
- blink a led one over two “cycles”.

\[
\text{node } \text{cpt(reset:bool) returns (sevseg: int; led_on: bool);}
\]
\[
\begin{align*}
\text{let} & \quad \text{sevseg} = 0 \rightarrow \text{if (reset or pre(sevseg = 9))} \\
& \quad \quad \text{then 0} \\
& \quad \quad \quad \text{else pre(sevseg)+1;} \\
& \quad \text{led_on} = \text{true} \rightarrow \text{not pre(led_on);} \\
\text{tel}
\end{align*}
\]
Running Example: Desired Result

Demo!
Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The Lustre language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?
Development Cycle for Lustre

✓ Write the synchronous program;

• Check program validity:
  • Causality (no cycle)
  • Initialisation analysis (pre)
  • Clock calculus ▶ No access to absent values.
  ▶ these checks are done at compile time

\[
\begin{align*}
  x &= y; \\
y &= x; \text{ -- forbidden}
\end{align*}
\]
Lustre in the Development Cycle

- Write the synchronous program;
- Check program validity
  - Compile → generation of C code
Sequential Code Generation - Goal

Generate a (C) program of the form:

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

```c
step()
done
```

▶ 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

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

compiles into (new init variable):

```pre
if init then dec = 0 else dec = i;
```
Compilation of our Example as a Simulation Loop

```plaintext
node cpt(reset:bool) returns
  (sevseg: int; led_on: bool);
let
  sevseg = 0  -> if (reset or pre(sevseg = 9))
    then 0
    else pre(sevseg)+1;
  led_on = true  -> not pre(led_on);
```

The compilation generates a `.c` and a main for simulation.

Demo (state + main).
Lustre in the Development Cycle

- 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.
- One more compilation: synchronous code + integration code ⇒ generates assembly (low abstraction level).
Running example: Complete Compilation Process

```
cpt.lus

lus2c

node.c
main.c

step()
```

```
while(1){}
```

```
extern cpt o led on()
extern cpt o sevseg()
extern cpt i ...
```

```
glue_arduino.c

#include arduino.h

cpt i getButton(){...}
cpt o led_sevseg(){...}
cpt o led_on(){...}
```

```
arduino.h

led_on()
led_off()
display7(int val)
```

```
cpt.elf
```

```
cpt.o

loop
```

```
node.c
main.c
```
Demo: glue code and interface.
Beyond Basic-Lustre

Around Lustre, a huge ecosystem:

- Formal verification, test, controller synthesis.
- Some work on data structures (arrays, iterators).
- Extensions with automata (Scade).
- A multithreaded implementation.
- Globally Asynchronous Locally Synchronous systems.
Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The Lustre language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?
Conclusion 1/2

- Lustre (resp. synchronous languages): niche language
  - success
- Before: the (specialized) programmers already wrote the same kinds of programs as specifications (but without proof or automatic code generation)
- After: clarity, safety is easier
  - engineer is “happy”. she has Scade

\(^2\)Esterel Technologies
Synchronous programming is not dead: numerous research topics around it!

- Hybrid (discrete / dense time) models
- WCET estimation
- Targetting multi-core platforms
- Certification of compilers
- Relaxing synchrony to address streaming applications
Plan

Motivation: Programming Safety Critical Embedded Systems

The Synchronous Approach

The Lustre language

Compilation of Synchronous Languages (Lustre)

Conclusion

What about [non-synchronous] dataflow?
Dataflow Applications Examples

Medical image processing [Albers2012]

Software Defined Radio [Dardaillon2014]

Video Decoding [Lucarz09]
Dataflow Applications Examples

- Medical image processing [Albers2012]
- Software Defined Radio [Dardaillon2014]
- Video Decoding [Lucarz09]
Dataflow Applications Examples

Medical image processing [Albers2012]

Software-Defined Radio

KPN ⟷ Lustre? ⟷ SDF

Video Decoding [Lucarz09]
Kahn Process Networks

- Processes communicate through unbounded FIFOs
- Non-blocking writes / Blocking reads → deterministic
- Necessary whenever you can’t predict production rates statically (eg compression)

---

3[Kahn1974]
Lustre vs KPN

Main differences

• Lustre’s flows are ... not buffers.
  • Just data dependencies

• If you need FIFO buffers, you need to encode them explicitly
  • Including synchronization

• This is error-prone and very inefficient
Lustre vs KPN

OK, but KPNs assume unbounded memory, so in practice ...

- ... Live with it and dimension your buffers “by experience” [RVC-Cal]
- ... Choose a trade-off, eg give clocks and have the type system dimension buffers for you, eg with Lucy-n [Plateau2010]
[Synchronous] Static DataFlow

- Restricted Kahn Processes
- Number of tokens produced/consumed is statically known
  ⇒ Statically schedulable

\[4\text{[Lee1987]}\]
Lustre vs SDF

Main differences

- Fifos need to be hand-coded in Lustre
- SDF doesn’t allow for absence of values (multi-clocked graphs)
  - Put differently: only multi-periodic designs can be constructed in SDF.
The End

Medical image processing [Albers2012]

Software Defined Radio

KPN Lustre? SDF

Video Decoding [Lucarz09]
Credits

- 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.


Bibliography III

*Systems*, CASES ’14, pages 8:1–8:10, New York, NY, USA, 2014. ACM.


