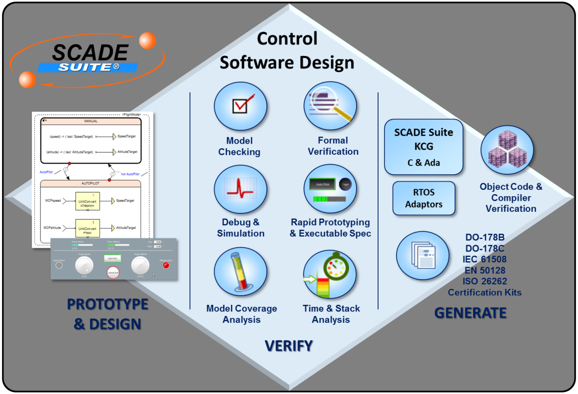
CS:5810 Formal Methods in Software Engineering

Reactive Systems and the Lustre Language
Part 2

Adrien Champion
adrien-champion@uiowa.edu

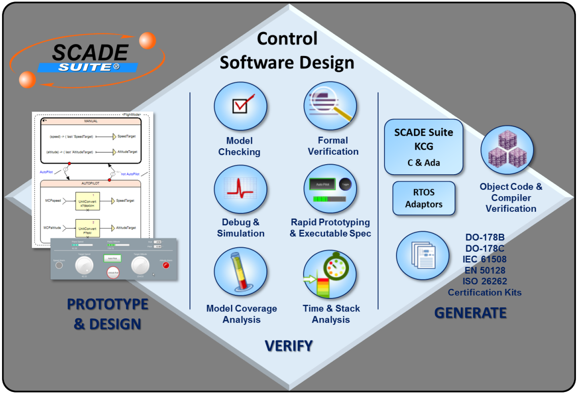
Embedded systems development



Embedded systems development

Pivot language between design and code should

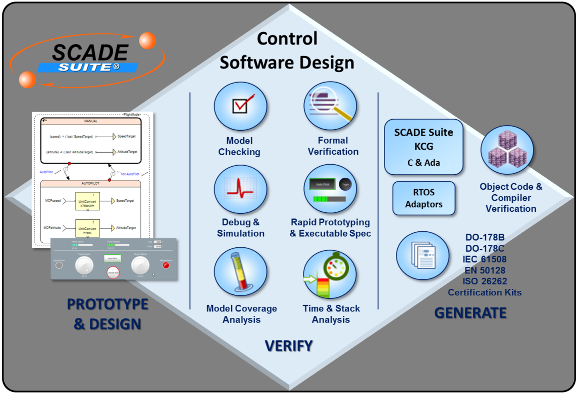
- have clear and precise semantics, and



Embedded systems development

Pivot language between design and code should

- have clear and precise semantics, and
- be consistent with design / prototype formats and target platforms



Lustre: a synchronous dataflow language

- Synchronous:

  a base clock regulates computations;

  computations are inherently parallel

- Dataflow:

  inputs, outputs, variables, constants ... are endless streams of values

Lustre: a synchronous dataflow language

- Synchronous:

  a base clock regulates computations;
  computations are inherently parallel

- Dataflow:

  inputs, outputs, variables, constants . . . are endless streams of values

- Declarative:

  set of equations, no statements

Lustre: a synchronous dataflow language

- Synchronous:

  a base clock regulates computations;

  computations are inherently parallel

- Dataflow:

  inputs, outputs, variables, constants ... are endless streams of values

- Declarative:

  set of equations, no statements

- Reactive systems:

  Lustre programs run forever
  At each clock tick they

  - compute outputs from their inputs
  - before the next clock tick

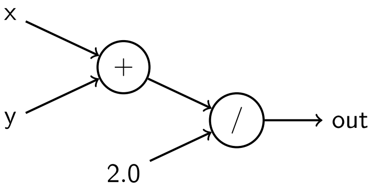
A simple example

```
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```

A simple example

```
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```

Circuit view:



A simple example

```
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```

Mathematical view:

$$\forall i \in \mathbb{N},\ \mathsf{out}_i = \frac{\mathsf{x}_i + \mathsf{y}_i}{2}$$

A simple example

```
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```

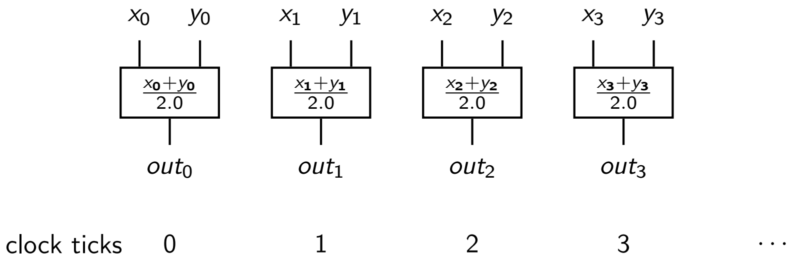
Transition system unrolled view:

clock ticks	0	1	2	3	$\cdots$

A simple example

```
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```

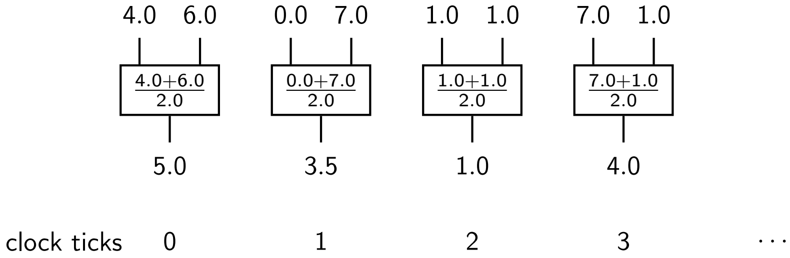
Transition system unrolled view:



A simple example

```
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```

Transition system unrolled view:



Combinational programs

- Basic type: `bool`, `int`, `real`
- Constants (i.e., constant streams):

2	2	2	2	2	2	...
`true`	`true`	`true`	`true`	`true`	`true`	...

Combinational programs

- Basic type: `bool`, `int`, `real`

- Constants (i.e., constant streams):

2	2	2	2	2	2	...
`true`	`true`	`true`	`true`	`true`	`true`	...

- Pointwise operators:

x	x_0	x_1	x_2	x_3	x_4	$\ldots$
y	y_0	y_1	y_2	y_3	y_4	$\ldots$
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$\ldots$

- All classical operators are provided

Combinational programs

Conditional expressions:

```
node max (n1,n2: real) returns (out: real);
let
  out = if (n1 >= n2) then n1 else n2;
tel
```

- Functional "`if ... then ... else ...`"
- It is an expression, **not a statement**

Combinational programs

Conditional expressions:

```
node max (n1,n2: real) returns (out: real);
let
  out = if (n1 >= n2) then n1 else n2;
tel
```

- Functional "`if ... then ... else ...`"
- It is an expression, **not a statement**

```
-- This does not compile
if (a >= b) then m = a else m = b;
```

Combinational programs

Local variables:

```
node max (a,b: real) returns (out: real);
var
  condition: bool;
let
  out = if condition then a else b;
  condition = a >= b;
tel
```

Combinational programs

Local variables:

```
node max (a,b: real) returns (out: real);
var
  condition: bool;
let
  out = if condition then a else b;
  condition = a >= b;
tel
```

- Order does not matter
- Set of equations not sequence of statements

Combinational programs

Local variables:

```
node max (a,b: real) returns (out: real);
var
  condition: bool;
let
  out = if condition then a else b;
  condition = a >= b;
tel
```

- Order does not matter
- Set of equations not sequence of statements
- Causality is resolved syntactically

Combinational programs

Combinational recursion is forbidden:

```
x = 1 / (2 - x);
```

Combinational programs

Combinational recursion is forbidden:

```
x = 1 / (2 - x);
```

- has a unique integer solution: $x = 1$,
- but is not computable step by step

Combinational programs

Combinational recursion is forbidden:

```
x = 1 / (2 - x);
```

- has a unique integer solution: $x = 1$,
- but is not computable step by step

Syntactic loop:

```
x = if c then y else 0;
y = if c then 1 else x;
```

Combinational programs

Combinational recursion is forbidden:

```
x = 1 / (2 - x);
```

- has a unique integer solution: $x = 1$,
- but is not computable step by step

Syntactic loop:

```
x = if c then y else 0;
y = if c then 1 else x;
```

- not a real (semantic) loop:

```
x = if c then 1 else 0;
y = x;
```

- but still forbidden by Lustre

Memory programs

- Previous operator " `pre` ":

  $(\texttt{pre}\ x)_0$ is undefined (`nil`)

  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

Memory programs

- Previous operator " `pre` ":

  $(\texttt{pre}\ x)_0$ is undefined (`nil`)

  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":

  $(x\ \texttt{->}\ y)_0 = x_0$

  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

Memory programs

- Previous operator " `pre` ":
  $(\texttt{pre}\ x)_0$ is undefined (`nil`)
  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":
  $(x\ \texttt{->}\ y)_0 = x_0$
  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

- Examples:

x	x_0	x_1	x_2	x_3	x_4	x_5	$\ldots$
`pre` x							

Memory programs

- Previous operator " `pre` ":
  $(\texttt{pre}\ x)_0$ is undefined (`nil`)
  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":
  $(x\ \texttt{->}\ y)_0 = x_0$
  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

- Examples:

x	x_0	x_1	x_2	x_3	x_4	x_5	...
`pre` x	`nil`	x_0	x_1	x_2	x_3	x_4	...

Memory programs

- Previous operator " `pre` ":

  $(\texttt{pre}\ x)_0$ is undefined (`nil`)

  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":

  $(x\ \texttt{->}\ y)_0 = x_0$

  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

- Examples:

x	x_0	x_1	x_2	x_3	x_4	x_5	...
`pre` x	`nil`	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
x `->` y							

Memory programs

- Previous operator " `pre` ":

  $(\texttt{pre}\ x)_0$ is undefined (`nil`)

  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":

  $(x\ \texttt{->}\ y)_0 = x_0$

  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

- Examples:

x	x_0	x_1	x_2	x_3	x_4	x_5	...
`pre` x	`nil`	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
x `->` y	x_0	y_1	y_2	y_3	y_4	y_5	...

Memory programs

- Previous operator " `pre` ":

  $(\texttt{pre}\ x)_0$ is undefined (`nil`)

  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":

  $(x\ \texttt{->}\ y)_0 = x_0$

  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

- Examples:

x	x_0	x_1	x_2	x_3	x_4	x_5	...
`pre` x	`nil`	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
x `->` y	x_0	y_1	y_2	y_3	y_4	y_5	...
2	2	2	2	2	2	2	...
2 `->` (`pre` x)							

Memory programs

- Previous operator " `pre` ":
  $(\texttt{pre}\ x)_0$ is undefined (`nil`)
  $(\texttt{pre}\ x)_i = x_{i-1}$ for $i > 0$

- Initialization " `->` ":
  $(x\ \texttt{->}\ y)_0 = x_0$
  $(x\ \texttt{->}\ y)_i = y_i$ for $i > 0$

- Examples:

x	x_0	x_1	x_2	x_3	x_4	x_5	$\dots$
`pre` x	`nil`	x_0	x_1	x_2	x_3	x_4	$\dots$
y	y_0	y_1	y_2	y_3	y_4	y_5	$\dots$
x `->` y	x_0	y_1	y_2	y_3	y_4	y_5	$\dots$
2	2	2	2	2	2	2	$\dots$
2 `->` (`pre` x)	2	x_0	x_1	x_2	x_3	x_4	$\dots$

Recursive definition using `pre` :

```
n = 0 -> 1 + pre n;
a = false -> not pre a;
```

n	0
a	false

Memory programs

Recursive definition using `pre` :

```
n = 0 -> 1 + pre n;
a = false -> not pre a;
```

n	0	1	2	3	...
a	false				

Memory programs

Recursive definition using `pre` :

```
n = 0 -> 1 + pre n;
a = false -> not pre a;
```

n	0	1	2	3	...
a	false	true	false	true	...

Memory programs: examples

```
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel
```

signal	0	1	1	0	1	0	...
e							

Memory programs: examples

```
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel
```

signal	0	1	1	0	1	0	...
e	0						

Memory programs: examples

```
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel
```

signal	0	1	1	0	1	0	...
e	0	1	0	0	1	0	...

Memory programs: examples

Raising edge:

```
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel
```

signal	0	1	1	0	1	0	...
e	0	1	0	0	1	0	...

Memory programs: examples

```
node guess (n: int) returns (out1,out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel
```

n	4	2	3	0	3	7	...
out1							

Memory programs: examples

```
node guess (n: int) returns (out1,out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel
```

n	4	2	3	0	3	7	...
out1	4						

Memory programs: examples

```
node guess (n: int) returns (out1,out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel
```

n	4	2	3	0	3	7	...
out1	4	2	2	0	0	0	...

Memory programs: examples

```
node guess (n: int) returns (out1,out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel
```

n	4	2	3	0	3	7	...
out1	4	2	2	0	0	0	...
out2	4	4	4	4	4	7	...

Memory programs: examples

Min and max of a sequence:

```
node guess (n: int) returns (out1,out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel
```

n	4	2	3	0	3	7	...
out1	4	2	2	0	0	0	...
out2	4	4	4	4	4	7	...

Exercises

Design a node

```
node switch (on,off: bool) returns (state: bool);
```

such that:

- `state` raises (false to true) if `on`;
- `state` falls (true to false) if `off`;

Exercises

Design a node

```
node switch (on,off: bool) returns (state: bool);
```

such that:

- `state` raises (false to true) if `on`;
- `state` falls (true to false) if `off`;
- everything behaves as if `state` was false at the origin;
- `switch` must work properly even if `on` and `off` are the same

Exercises

Design a node

```
node switch (on,off: bool) returns (state: bool);
```

such that:

- state raises (false to true) if on;
- state falls (true to false) if off;
- everything behaves as if state was false at the origin;
- switch must work properly even if on and off are the same

```
node switch (on, off: bool) returns (state: bool);
let
  state =
    false -> if (not pre state) then on
                                else (not off);
    -- Equivalently:
    --         ((not pre state) and on)
    --     or ((pre state) and (not off))
tel
```

Exercises

Compute the sequence $1,\ 1,\ 2,\ 3,\ 5,\ 8\ \ldots$

Exercises

Compute the sequence 1, 1, 2, 3, 5, 8, 13, 21 ...

Fibonacci sequence:

$u_0 = u_1 = 1$

$u_n = u_{n-1} + u_{n-2} \quad \text{for } n \geq 2$

Credits

These notes are based on the following lectures notes:

The Lustre Language — Synchronous Programming
by Pascal Raymond and Nicolas Halbwachs
Verimag-CNRS