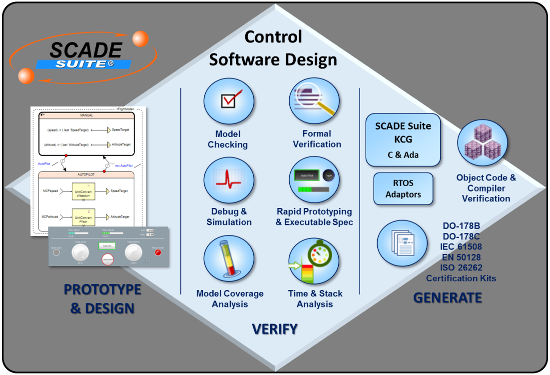
CS:5810 Formal Methods in Software Engineering

Reactive Systems and the Lustre Language[1]

Adrien Champion Cesare Tinelli

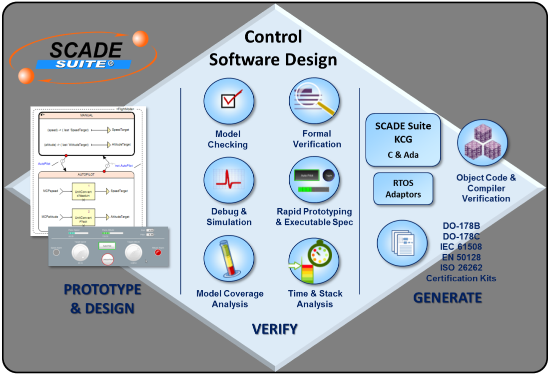
Embedded systems development



Embedded systems development

Intermediate modeling language between design and code should

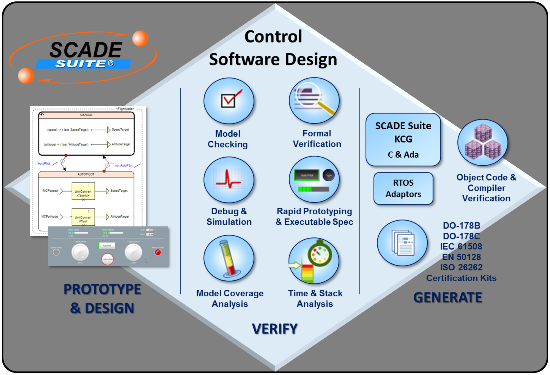
- have clear and precise semantics, and



Embedded systems development

Intermediate modeling 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 current inputs and state
  - before the next clock tick

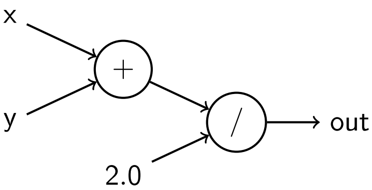
A simple example

```
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
```

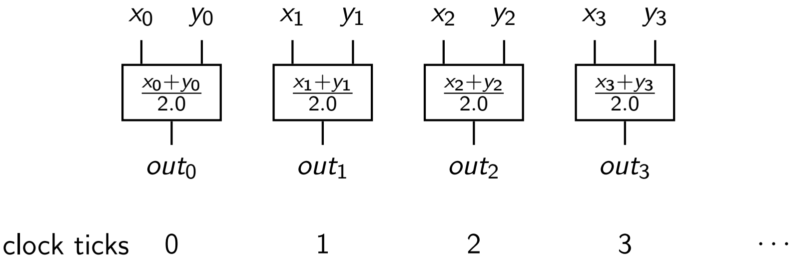
Transition system unrolled view:

clock ticks	0	1	2	3	···

A simple example

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

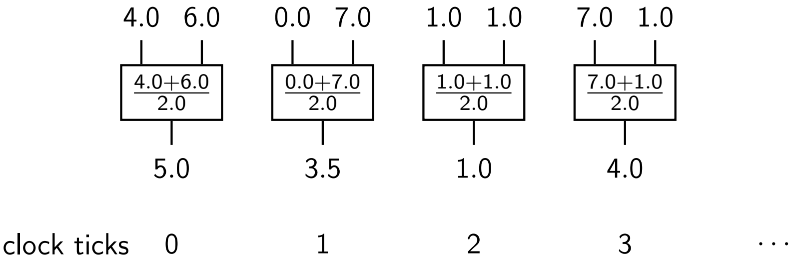
Transition system unrolled view:



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 types: `bool`, `int`, `real`

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

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

- Basic types: `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	$\dots$
y	y_0	y_1	y_2	y_3	y_4	$\dots$
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$\dots$

- 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
  cond: bool;
let
  out = if cond then a else b;
  cond = (a >= b);
tel
```

Combinational programs

Local variables:

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

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

Combinational programs

Combinational recursion is forbidden:

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

- the equation above has a unique integer solution: $x = 1$,
- but it 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);
```

- the equation above has a unique integer solution: $x = 1$,
- but it 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

Stateful programs

Previous operator `pre` :

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

Stateful programs

Previous operator `pre` :

(`pre` $x)_0$ is undefined (`nil`)

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

Initialization `->` :

$(x$ `->` $y)_0 = x_0$

$(x$ `->` $y)_i = y_i$ for $i > 0$

Stateful 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							

Stateful 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	//	x_0	x_1	x_2	x_3	x_4	...

Stateful 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	...
$\texttt{pre}\ x$	//	x_0	x_1	x_2	x_3	x_4	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
$x\ \texttt{->}\ y$							

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	//	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	...

Stateful 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	//	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)							

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	//	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)	2	x_0	x_1	x_2	x_3	x_4	...

Stateful programs

Recursive definitions using `pre` :

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

n	0
a	false

Stateful programs

Recursive definitions using `pre`:

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

n	0	1	2	3	...
a	false				

Stateful programs

Recursive definitions using `pre` :

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

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

Stateful programs: examples

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

signal	false	true	true	false	true	false	...
e							

Stateful programs: examples

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

signal	false	true	true	false	true	false	...
e	false						

Stateful programs: examples

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

signal	false	true	true	false	true	false	...
e	false	true	false	false	true	false	...

Raising edge:

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

signal	false	true	true	false	true	false	...
e	false	true	false	false	true	false	...

Stateful programs: examples

```
node guess (n: int) returns (o1, o2: int);
let
  o1 = n -> if (n < pre o1) then n else pre o1;
  o2 = n -> if (n > pre o2) then n else pre o2;
tel
```

n	4	2	3	0	3	7	...
o1							

Stateful programs: examples

```
node guess (n: int) returns (o1, o2: int);
let
  o1 = n -> if (n < pre o1) then n else pre o1;
  o2 = n -> if (n > pre o2) then n else pre o2;
tel
```

n	4	2	3	0	3	7	...
o1	4						

Stateful programs: examples

```
node guess (n: int) returns (o1, o2: int);
let
  o1 = n -> if (n < pre o1) then n else pre o1;
  o2 = n -> if (n > pre o2) then n else pre o2;
tel
```

n	4	2	3	0	3	7	...
o1	4	2	2	0	0	0	...

Stateful programs: examples

```
node guess (n: int) returns (o1, o2: int);
let
  o1 = n -> if (n < pre o1) then n else pre o1;
  o2 = n -> if (n > pre o2) then n else pre o2;
tel
```

n	4	2	3	0	3	7	...
o1	4	2	2	0	0	0	...
o2	4	4	4	4	4	7	...

Min and max of a sequence:

```
node guess (n: int) returns (o1, o2: int);
let
  o1 = n -> if (n < pre o1) then n else pre o1;
  o2 = n -> if (n > pre o2) then n else pre o2;
tel
```

n	4	2	3	0	3	7	...
o1	4	2	2	0	0	0	...
o2	4	4	4	4	4	7	...

Design a node

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

such that:

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

Design a node

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

such that:

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

Exercises

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

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

Fibonacci sequence:

$u_0 = u_1 = 1$

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