Embedded Systems Development

Lecture 5
Esterel, LUSTRE & SCADE

Daniel Kästner
AbsInt Angewandte Informatik GmbH
kaestner@absint.com
Trap Level Propagation

\[ \text{Must}(\uparrow^m \text{trap } T \text{ in } p \text{ end}, \overline{E}) = \text{Must}(\{\uparrow^{m+1} \text{ } p\}^T, \overline{E}) \]

\[ \text{Must}(\{\uparrow^m q\}^T, \overline{E}) = \left\langle \text{Must}_s(\uparrow^m q, \overline{E}'), \downarrow \text{Must}_k(\uparrow^m q, \overline{E}') \right\rangle \]

where \( \overline{E}' = (E, \tau') \) and \( \tau'(T) = m \)

\[ \text{Must}(\uparrow^m q, \overline{E}) = \left\langle \text{Must}_s(q, \overline{E}), \uparrow^m \text{ Must}_k(q, \overline{E}) \right\rangle \]

\[ \text{Must}(\uparrow^m \text{ exit } T, \overline{E}) = \left\langle \emptyset, \text{Must}_k(\uparrow^m \text{ exit } T, \overline{E}) \right\rangle \]

\[ \text{Must}_k(\uparrow^m \text{ exit } T, \overline{E}) = 2 + m - \tau(T) \]

Intuitive completion code rule:

- exit \( T \) is encoded by 2, if the directly enclosing trap declaration is that of \( T \), and \( n + 2 \) if \( n \) trap declarations have to be traversed before reaching that of \( T \).

\[ \text{Must}_k(\text{trap } T \text{ in exit } T \text{ end}, \overline{E}) = \text{Must}_k(\{\uparrow \text{ exit } T\}^T, \overline{E}) \]

\[ = \text{Must}_k(\{2 + 1 - 1\}^T, \overline{E}) = \text{Must}_k(\{2\}^T, \overline{E}) = \downarrow 2 = 0 \]
Trap Level Propagation

\[ \text{Can}^x(\uparrow^m \text{trap } T \text{ in } p \text{ end}, \overline{E}) = \text{Can}^x(\{\uparrow^m + 1 \ p\}^T, \overline{E}) \]
\[ \text{Can}^x(\{\uparrow^m q\}^T, \overline{E}) = \left\langle \text{Can}^x_S(\uparrow^m q, \overline{E}'), \downarrow \text{Can}^x_k(\uparrow^m q, \overline{E}') \right\rangle \]
where \( \overline{E}' = (E, \tau') \) and \( \tau'(T) = m \)
\[ \text{Can}^x(\uparrow^m q, \overline{E}) = \left\langle \text{Can}^x_S(q, \overline{E}), \uparrow^m \text{Can}^x_k(q, \overline{E}) \right\rangle \]
\[ \text{Can}^x(\uparrow^m \text{ exit } T, \overline{E}) = \left\langle \emptyset, \text{Can}^x_k(\uparrow^m \text{ exit } T, \overline{E}) \right\rangle \]
\[ \text{Can}^x_k(\uparrow^m \text{ exit } T, \overline{E}) = 2 + m - \tau(T) \]

\[ \downarrow k = \begin{cases} 
0, & \text{if } k = 0 \text{ or } k = 2 \\
1, & \text{if } k = 1 \\
k - 1, & \text{if } k > 2 
\end{cases} \]
\[ \uparrow k = \begin{cases} 
k, & \text{if } k = 0 \text{ or } k = 1 \\
k + 1, & \text{if } k \geq 2 
\end{cases} \]
Constructive Causality: Example

module awaitImmediate:
  input S;
  output O;
  trap T in
    loop
      present S
      then exit T
      else pause
    end present
  end loop
end trap
emit O;
end module
Advanced Constructiveness

- Preemption statements behave as tests for the guard in each instant where the guard is active.

module Py:
output O;
abort
  sustain O
when O

constructive in the first instant
non-constructive (non reactive) in later instants
- Now for something completely different...
Program Analysis – An Outlook

- Analysis for constructive causality (Must and Can predicates) is a program analysis, more precisely dataflow analysis at the Esterel level.
- Program analyses can be defined for any programming language, including, e.g., processor instruction sets.
- Observation: Cache analysis at the executable level is very similar to analysis for constructive causality:
  - Must Analysis:
    For each program point and calling context, find out which blocks must be in the cache → cache hit.
  - May Analysis:
    For each program point and calling context, find out which blocks may be in the cache → complement: cache miss.
- General semantics-based framework for program analysis: Abstract Interpretation.
Program Analysis – An Outlook

Example:
Fully associative data cache (2 elements) with Least Recently Used (LRU) replacement

Load of data at memory address $a$

- $a \rightarrow a$ (hit)
- $b \rightarrow b$ (miss)
- $b \rightarrow a$ (miss)
- $a \rightarrow a$ (hit)
- $a \rightarrow b$ (hit)
- $d \rightarrow d$ (hit)
- $d \rightarrow a$ (hit)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow a$ (hit)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)

Must: $a$
May: $a, d, b$

Example:
Fully associative data cache (2 elements) with Least Recently Used (LRU) replacement

Load of data at memory address $a$

- $a \rightarrow a$ (hit)
- $b \rightarrow b$ (miss)
- $b \rightarrow a$ (miss)
- $a \rightarrow a$ (hit)
- $a \rightarrow b$ (hit)
- $d \rightarrow d$ (hit)
- $d \rightarrow a$ (hit)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)

Must: $a$
May: $a, d, b$

Example:
Fully associative data cache (2 elements) with Least Recently Used (LRU) replacement

Load of data at memory address $a$

- $a \rightarrow a$ (hit)
- $b \rightarrow b$ (miss)
- $b \rightarrow a$ (miss)
- $a \rightarrow a$ (hit)
- $a \rightarrow b$ (hit)
- $d \rightarrow d$ (hit)
- $d \rightarrow a$ (hit)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)

Must: $a$
May: $a, d, b$

Example:
Fully associative data cache (2 elements) with Least Recently Used (LRU) replacement

Load of data at memory address $a$

- $a \rightarrow a$ (hit)
- $b \rightarrow b$ (miss)
- $b \rightarrow a$ (miss)
- $a \rightarrow a$ (hit)
- $a \rightarrow b$ (hit)
- $d \rightarrow d$ (hit)
- $d \rightarrow a$ (hit)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)
- $d \rightarrow a$ (miss)
- $a \rightarrow d$ (hit)

Must: $a$
May: $a, d, b$
- ... and back again.
Model-based Software Development

- Esterel programs
  - SCADE language
  - SyncCharts

- aiT WCET Analyzer
  - Timing Validation

- SymTA/S
  - System-level
  - Schedulability Analysis

- Generator
- Compiler
- C Code
- Binary Code
LUSTRE

- Programs are structured into nodes:
  - Node: subprogram defining its output parameters as functions of its input parameters.
  - Definition given by unordered set of equations ($\rightarrow$ declarative language)
- Based on synchronous data-flow model:
  - Functional: no side effects.
  - All nodes work simultaneously, ie at the same speed.
  - No broadcasting of signals; sequencing and synchronization only from data dependences.
  - Each variable takes a value at every cycle of the program.

\[
\begin{align*}
  x & \rightarrow 2 \\
  y & \rightarrow + \\
  \ast & \rightarrow s \\
\end{align*}
\]

At any cycle $n$:
\[
    s_n = 2(x_n + y_n)
\]

- Basis of SCADE.
- Example:

```plaintext
node Counter (init, incr: int; reset: bool)
  returns (count:int);
let
    count = if reset then init
            else pre(count)+incr;
tel
```
Required Properties

- **Causality:** The output at any instant $t$ may only depend upon input received before or at $t$.

- **Bounded memory:** There must be a finite bound such that, at each instant the number of past input values necessary to produce a new output value remains smaller than that bound.

- **Efficient** code generation.

- Execution time **predictability:** no unbounded loops, no recursion.
Flows and Clocks

- Any variable and expression denotes a flow, i.e., a pair made \((x, b_x)\) of
  - a possibly infinite sequence \(X\) of values of a given type
  - a clock \(b_x\), representing a sequence of times.
- \(x\) is defined at instant \(i\) iff \(b_x(i) = true\).
- A flow takes its \(n\)-th value in the \(n\)-th time of its clock.
- Input variables are defined at every instant: their clock is called the basic clock.
- Example: Let \(x\) run on the basic clock \(C\), \(y\) on a slower clock. This gives the following time scales:

<table>
<thead>
<tr>
<th>Basic time-scale</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_x)</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>(x) time-scale</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>(b_y)</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>(y) time-scale</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Types, Equations, Assertions

- Variables are declared with their type:
  - Basic types: boolean, integer, real.
  - Type constructor tuple.
    - Semantics is Cartesian product.
  - Abstract types via import (cf. Esterel).

- Equations:
  - Variables are defined via equations, e.g. \( X = E \) with variable \( X \) and expression \( E \).
  - **Substitution** principle: \( X \) can be substituted to \( E \) anywhere in the program and vice versa.
  - **Definition** principle: The behavior of \( X \) must be completely specified by this equation.

- Assertions:
  - Assertions \( \text{assert}(E) \): \( E \) must hold during execution.
  - Used to optimize code generation, for simulation and for verification.
Variables and Expressions

- Operators only operate on operands sharing the same clock.
- As variables and expressions are streams, operators also produce streams. Example: With $x = (0,1,2,3,4,...)$ and $y = (2,4,6,8,10,...)$: $x+y=(2,5,8,11,14,...)$
- Expressions are build from variables, constants and operators.
- Three types of operators:
  - Data operators:
    - arithmetic, boolean and relational expressions
    - conditional expressions: $if \ E \ then \ X \ else \ Y$
  - Imported operators:
    - functions imported from host language
  - Temporal (sequence) operators.
Temporal Operators

- 'previous' operator \texttt{pre}:
  - \((\text{pre}(E))_0 = \bot\) (undefined, also denoted nil)
  - \((\text{pre}(E))_n = E_{n-1}\)
- 'followed by' operator \texttt{->}
  - \((E->F)_0 = E_0\)
  - \((E->F)_n = F_n\)
- (Down-)Sampling: \texttt{when}
  - Let \(E\) be an expression and \(B\) a boolean expression with the same clock: \((E \text{ when } B)\) is the sequence of values of \(E\) when \(B\) is true.
- Upsampling/Interpolation/Projection: \texttt{current}
  - Let \(E\) be an expression and \(B\) a boolean expression defining the clock of \(E\): Then \(\text{current } E\) has the same clock as \(B\); and \((\text{current } E)\) is the sequence of values of \(E\) at the last time when \(B\) was true.
Example

<table>
<thead>
<tr>
<th>B</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_7$</td>
<td>$x_8$</td>
</tr>
<tr>
<td>Y = X when B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z = current Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Example

<table>
<thead>
<tr>
<th>B</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_7$</td>
<td>$x_8$</td>
</tr>
<tr>
<td>Y = X when B</td>
<td>$x_2$</td>
<td>$x_4$</td>
<td></td>
<td></td>
<td>$x_7$</td>
<td>$x_8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z = current Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: `gaps` are not filled.
### Example

<table>
<thead>
<tr>
<th>$B$</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>$x_6$</td>
<td>$x_7$</td>
<td>$x_8$</td>
</tr>
<tr>
<td>$Y = X$ when $B$</td>
<td></td>
<td>$x_2$</td>
<td></td>
<td>$x_4$</td>
<td></td>
<td></td>
<td>$x_7$</td>
<td>$x_8$</td>
</tr>
<tr>
<td>$Z = \text{current } Y$</td>
<td>$\perp$</td>
<td>$x_2$</td>
<td>$x_2$</td>
<td>$x_4$</td>
<td>$x_4$</td>
<td>$x_4$</td>
<td>$x_7$</td>
<td>$x_8$</td>
</tr>
</tbody>
</table>
Clock Rules

- Let a clock environment $\omega$ be a function from identifiers to clocks.
- Let $CK(E, \omega)$ be the clock of the expression $E$ in the environment $\omega$.
- For an equation $X=E$ holds $\omega(X)=CK(E, \omega)$.
- Let $\perp$ be the undefined clock and $T$ the erroneous clock. Then
  $$ck \leq ck' \Leftrightarrow (ck = \perp \lor ck' = T \lor ck \equiv ck')$$
- Let $\cup$ denote the least upper bound operator.
- Constants: For any constant $k$, $CK(k, \omega)=true$ (the basic clock).
- Variables: For any identifier $X$, $CK(X, \omega)=\omega(X)$.
- Synchronous operators:
  $$CK(op(E_1, E_2, ..., E_n), \omega) = \bigcup_{i=1}^{n} CK(E_i, \omega)$$
Clock Rules

- **Downsampling**: The operands of the `when` operator must be on the same clock: $CK(E \ when\ ck, \omega)=ck$.
- **Upsampling**: Let $ck$ be the clock of $E$, $ck \neq true$: $CK(current(E), \omega)=ck$.
- Clock of a node instance: clock of its effective inputs.
- Initialization problem: $current(X \ when\ C)$ exists but is undefined ($\bot$) until $C$ becomes $true$ for the first time.
- Solution: activation conditions
  - Not an operator, rather a macro.
  - $x = CONDUCT(OP, \ clk, \ args, \ dflt)$ equivalent to $X = if \ clk \ then \ current(OP(args \ when \ clk)) else (dflt \ -> \ pre(X))$
  - Provided by SCADE (not part of LUSTRE).
### Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ n = (0 \rightarrow \text{pre}(n) + 1) \]

\[ e = (1 \rightarrow \text{not pre}(e)) \]

\[ \text{n when e} \]

\[ \text{current(n when e)} \]

\[ \text{Counter((1,1,false) when C)} \]

\[ \text{Counter(1,1,false) when C} \]

```plaintext
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
               else \text{pre(count)} + incr;
tel
```
Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (1-&gt;not pre(e))</td>
<td>n when e</td>
<td>current(n when e)</td>
<td>Counter((1,1,false) when C)</td>
<td>Counter(1,1,false) when C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

``` cider
node Counter (init: int; incr: int; reset: bool) 
  returns (count: int);
let 
  count = init -> if reset then init 
  else pre(count)+incr;
tel
```
Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (1-&gt;not pre(e))</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

n when e

current(n when e)

Counter((1,1,false) when C)

Counter(1,1,false) when C

```
node Counter (init, incr: int; reset: bool)
  returns (count:int);
  let
  
  count = init -> if reset then init
  
  else pre(count)+incr;

tel
```
Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (1-&gt;not pre(e))</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>2</td>
<td></td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>current(n when e)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter(1,1,false) when C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
    else pre(count)+incr;
tel
## Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (1-&gt;not pre(e))</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current(n when e)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
node Counter (init, incr: int; reset: bool)
    returns (count: int);
let
    count = init -> if reset then init
    else pre(count)+incr;
tel
```
## Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (1-&gt;not pre(e))</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>current(n when e)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter(1,1,false) when C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```latex
node Counter (init, incr: int; reset: bool)
    returns (count:int);
let
    count = init -> if reset then init
        else pre(count)+incr;
tel
```
### Example

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n=(0-&gt;pre(n)+1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>e = (1-&gt;not pre(e))</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n when e</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>current(n when e)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Counter((1,1,false) when C)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter(1,1,false) when C</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```node
Counter (init, incr: int; reset: bool)
returns (count:int);
let
count = init -> if reset then init
else pre(count)+incr;
tel
```
Program Structure

- **Nodes** are LUSTRE subprograms. General structure:

  \[
  \text{node } N \left( x_1: \tau_1; x_2: \tau_2; \ldots; x_p: \tau_p \right) \\
  \text{returns } \left( y_1: \theta_1; y_2: \theta_2; \ldots; y_p: \theta_q \right) \\
  \text{var } z_1: \gamma_1; z_2: \gamma_2; \ldots; z_k: \gamma_k \\
  \text{let} \\
  \quad z_1 = E_1; \ldots; z_k = E_i; \\
  \quad y_1 = E_j; \ldots; y_p = E_m; \\
  \text{tel}
  \]

- **Node instantiation**: if \( N \) is the name of a node with above signature and if \( E_1, \ldots, E_p \) are expressions of type \( \tau_1, \ldots, \tau_p \), then \( N(E_1, \ldots, E_p) \) is an expression of type \( \text{tuple}(\theta_1, \ldots, \theta_q) \).

- Conditional and sequence operators are polymorphic and can be applied to tuples.
Arrays and Recursion

- Let \( n \) be a constant. Given type \( \tau \), \( \tau_n \) defines an array with \( n \) entries of type \( \tau \).
- Example: \( x:bool_n \)
- The bounds of an array must be known at compile time; the compiler transforms an array of \( n \) values into \( n \) different variables.
- \( X[i] \) denotes \( i \)th element.
- \( X[i..j] \) denotes the array made of elements \( i \) to \( j \) of \( X \).

- LUSTRE only allows static recursion: the recursion is completely unrolled.
- Attention: if the recursion is not bounded the compiler will not stop.
Compilation of LUSTRE Programs

- Static compiler checks:
  - Definition checking: any local and output variable must have exactly one definition.
  - No recursive node calls.
  - Clock consistency.
  - Absence of uninitialized expressions (yielding ⊥).
  - Absence of cyclic definitions.

- Compilation to
  - single-loop code
  - automata code.
Causality Problems

- LUSTRE only allows acyclic equation systems. Note: acyclic equations have a unique solution.
- $X = E$ is acyclic if $X$ does not occur in $E$ unless as subterm of the $pre$ operator.
- Examples:
  - $X = X$ and $pre(X)$ is cyclic
  - $X = Y$ and $pre(X)$ is acyclic

- Also structural deadlocks which are not true ones are rejected:
  - $X = if \ C \ then \ Y \ else \ Z$
  - $Y = if \ C \ then \ Z \ else \ X$
- Improved causality analysis in SCADE.
Clock Consistency

- Consider the following (illegal) example:
  \[
  b = \text{true} -> \text{not pre b;}
  y = x + (x \text{ when b});
  \]

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(x_0)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(x \text{ when b})</td>
<td>(x_0)</td>
<td></td>
<td>(x_2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x + (x \text{ when b}))</td>
<td>(x_0 + x_0)</td>
<td>(x_1 + x_2)</td>
<td>(x_2 + x_4)</td>
<td>(x_3 + x_6)</td>
<td></td>
</tr>
</tbody>
</table>

- The computation of the \(2n\)th value of \(y\) needs the \(2n\)th and the \(n\)th values of \(x\).
- Problem: not possible with bounded memory.
- Consequence: only streams of the same clock can be combined.
- Problem: undecidable whether two boolean expressions denote the same flow.
Clock Consistency (c’ed)

- Thus: two boolean expressions define the same clock iff they can be **syntactically** unified.
- Examples:
  \[ x = a \text{ when } (y \gt z) \]
  \[ y = b + c \]
  \[ u = d \text{ when } (b + c \gt z) \]
  \[ v = e \text{ when } (z \lt y) \]

- \( x \) and \( u \) share the same clock.
- \( x \) and \( v \) have different clocks.
Node Expansion

- **No modular compilation**: the code for a node may depend on its context.
- **Example**:
  
  ```
  node two_copies(a,b: int) returns (x,y: int);
  let x=a; y=b; end
  ```

- Two possible sequential codes:
  - `x:=a; y:=b`
  - `y:=b; x:=a`

- For the call `(x,y)=two_copies(a,x)`, only the first variant is correct.

- Thus: all nodes have to be expanded before compilation.
  - Formal parameters are substituted with actual ones.
  - Local variables are given unique names.
  - Called node body is inserted into the calling node body.