

# Translating Discrete-time Simulink to Lustre

Reference: *Translating Discrete-time Simulink to Lustre*. Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis. *EMSOFT03*



# Our view: a development process in three layers

Simulink/Stateflow

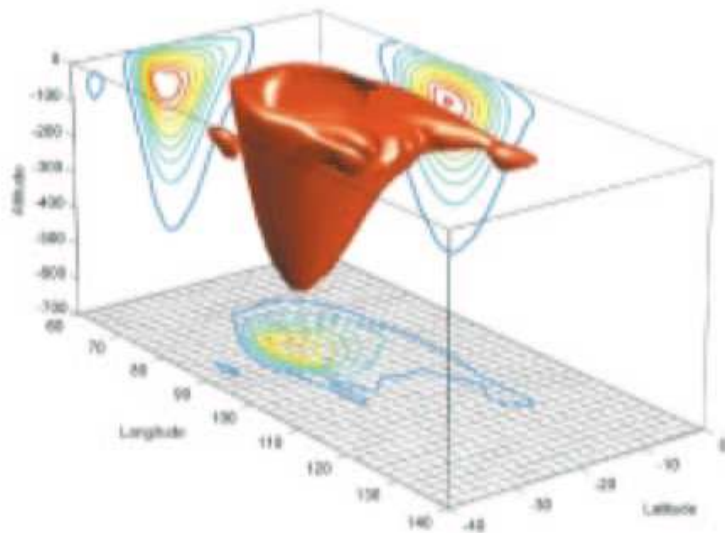
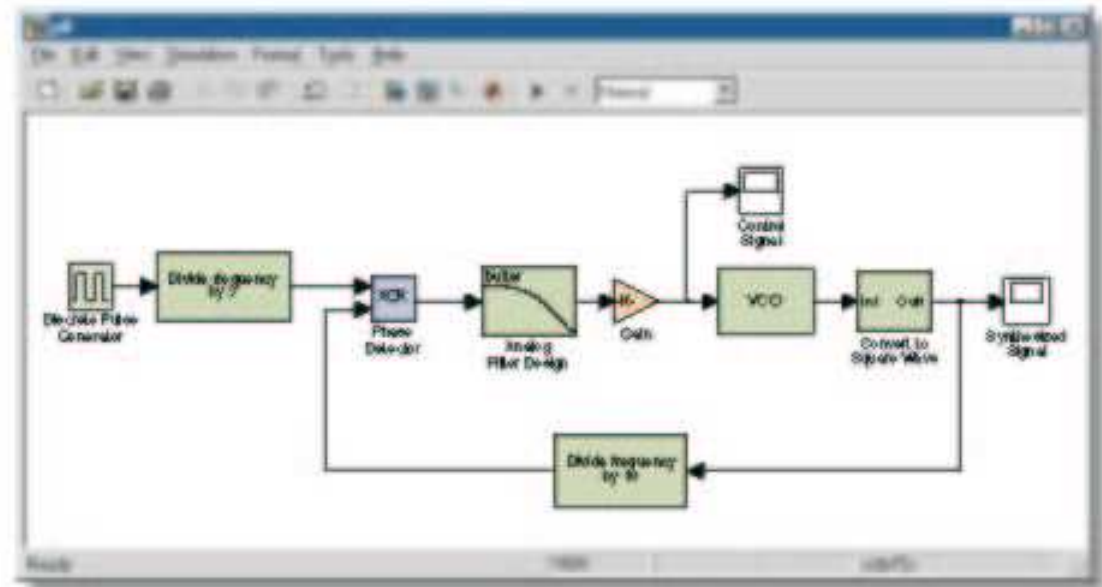
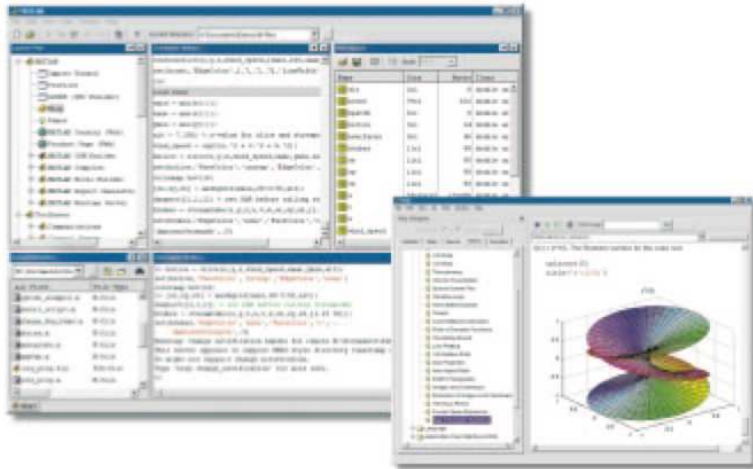
Lustre/SCADE

TTA

# Why Simulink/Stateflow ?

- De-facto standard in automotive industry.
- Good-looking GUI and simulator.
- Lots of plug-ins (applied math and control libraries, ...)
- [www.mathworks.com](http://www.mathworks.com)

# Matlab/Simulink/Stateflow



The Instrument Control Toolbox allows you to communicate with instruments directly from MATLAB. Here, a waveform is read from an oscilloscope into MATLAB and then plotted.

# Why Lustre ?

- Clean, simple, formal semantics.
- Analysis tools: simulators, model-checkers, test generators, theorem-provers, ...
- Programming-oriented: strong typing system, optimized code generators, ...
- Commercial version (SCADE)
  - DO178A certified.
  - Well-accepted in avionics industry (Airbus).
- [www-verimag.imag.fr](http://www-verimag.imag.fr)

# The development process

1. **Design** your controller in Simulink/Stateflow.
2. **Translate** it to Lustre.
3. **Verify** the Lustre program (transparent).
4. **Distribute** the Lustre program on TTA.
5. **Generate** code, compile and run.

Simulink/Stateflow

Lustre/SCADE

TTA

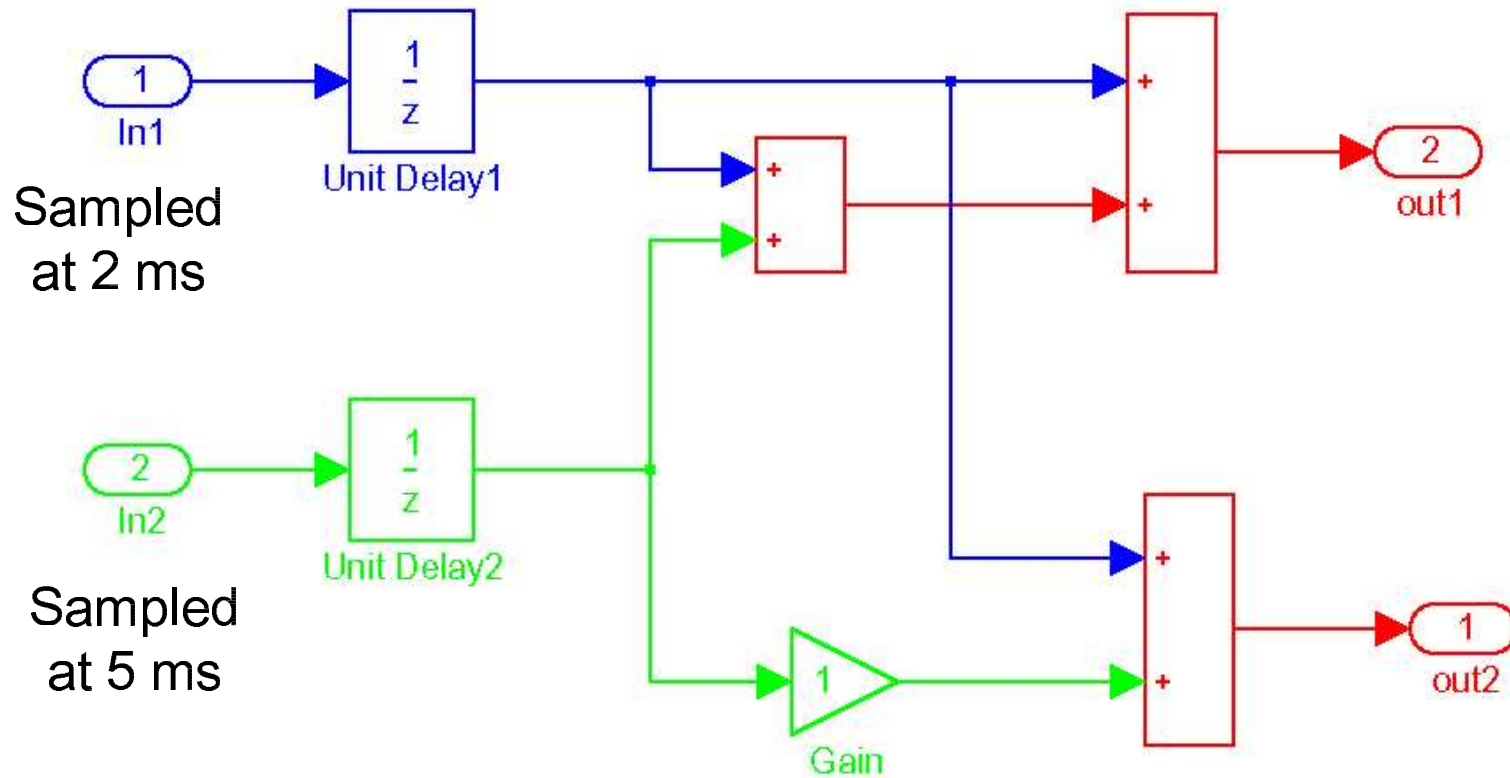
**Goal:  
Automatic  
as much as  
possible!**

# Translating Simulink to Lustre

- Both are **dataflow** languages: signals and systems.
- Simulink has both **continuous-time** and **discrete-time** blocks.
- We only translate **discrete-time** Simulink: **the controller**.
- **Goal: preserve semantics of Simulink.**

What semantics?

# A strange Simulink model



Simulink rejects the model with the Gain.  
It accepts the model if the Gain is removed !

# Simulink versus Lustre

Discrete-time Simulink

Lustre

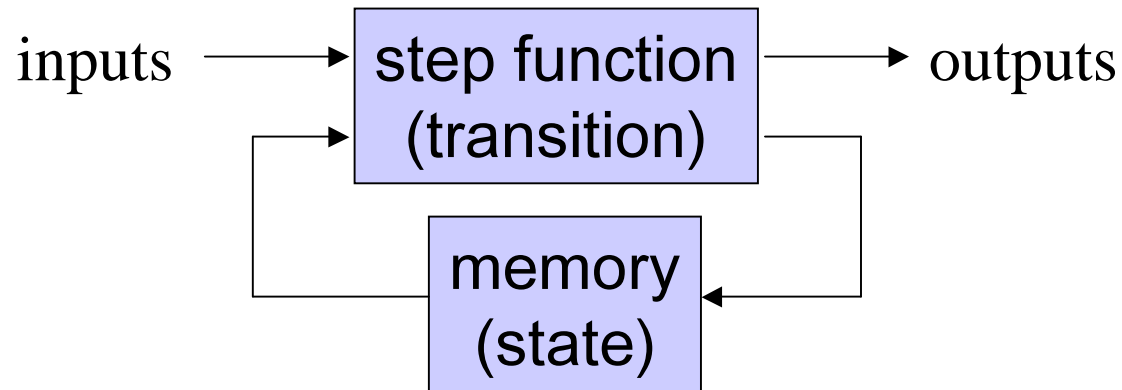
Piece-wise constant continuous signals	“Logical” time
No formal semantics ( <i>also depends on user options</i> )	Formal semantics (synchronous)
No strong typing, no time modularity, etc.	Strong typing, clock modularity, etc.

# Plan

- A glance into Lustre
- Translation:
  - Type inference
  - Clock inference
  - Bottom-up hierarchical translation

# A glance into Lustre

- A Lustre program models a Mealy machine:



- Implementing a Lustre program is easy:
  - Read inputs;
  - Compute next state and outputs;
  - Write outputs;
  - Update state;
  - Repeat at every clock “tick” (external event).

# A glance into Lustre

- A simple Lustre program:

```
node Counter2() returns(x:int);  
let  
  x = 0 -> pre(x) + 2;  
tel
```

- No inputs
- Output: x
- State: pre(x) (previous value of x)

# A glance into Lustre

- A simple Lustre program:

```
node Counter2() returns(x:int);  
let  
  x = 0 -> pre(x) + 2;  
tel
```

- $x$  is a logical-time (discrete-time) signal:

Time: $k$	0	1	2	...
$x(k)$	0	2	4	...

# A glance into Lustre

- Multi-clocked systems:

```
x = 0 -> pre(x) + 2;
```

```
b = true -> not pre(b);
```

```
y = x when b;
```

```
z = current(y);
```

time	0	1	2	3	4	...
x	0	2	4	6	8	...
b	true	false	true	false	true	...
y	0		4		8	...
z	0	0	4	4	8	...

# Translation steps

1. Type inference
2. Clock inference
3. Bottom-up block-by-block translation

# Translation steps

1. Type inference
2. Clock inference
3. Bottom-up block-by-block translation

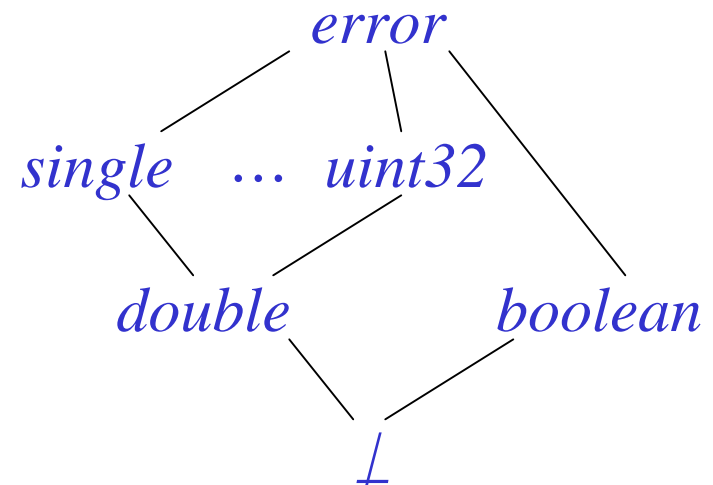
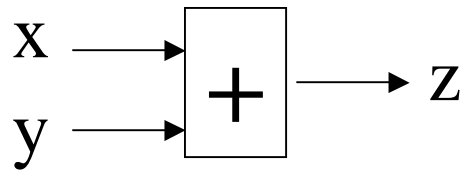
# Simulink types

- Types are **not mandatory** in Simulink.
- Available types: *double, single, int32, int16, int8, ..., boolean*.
- By default signals are “*double*”.
- ...except when blocks say otherwise:

<i>Constant</i> <sub><math>\alpha</math></sub>	$\alpha, \alpha \in SimNum$
<i>Adder</i>	$\alpha \times \dots \times \alpha \rightarrow \alpha, \alpha \in SimNum$
<i>Relation</i>	$\alpha \times \alpha \rightarrow boolean, \alpha \in SimNum$
<i>Logical Operator</i>	$boolean \times \dots \times boolean \rightarrow boolean$
<i>Disc. Transf. Func.</i>	$double \rightarrow double$
<i>Data Type Conv</i> <sub><math>\alpha</math></sub>	$\beta \rightarrow \alpha, \alpha, \beta \in SimNum$

# Simulink type inference

- Fix-point computation on a lattice:
- E.g.:



## Fix-point equations:

$$tx = \text{sup}(\text{double}, tx, ty, tz)$$

$$ty = \text{sup}(\text{double}, tx, ty, tz)$$

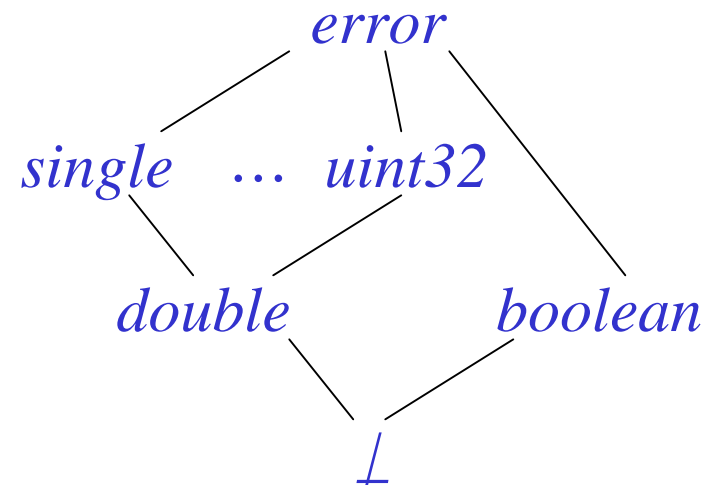
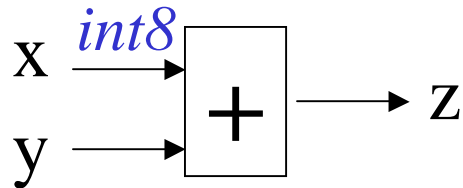
$$tz = \text{sup}(\text{double}, tx, ty, tz)$$

## Least fix-point:

$$tx = ty = tz = \text{double}$$

# Simulink type inference

- Fix-point computation on a lattice:
- E.g.:



## Fix-point equations:

$$tx = \text{int8}$$

$$ty = \text{sup}(\text{double}, tx, ty, tz)$$

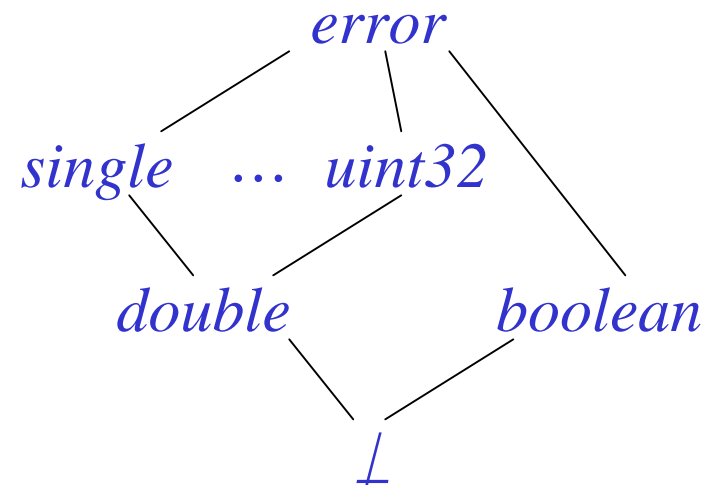
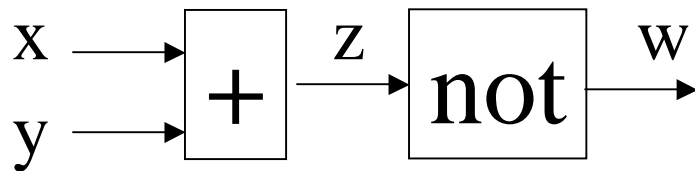
$$tz = \text{sup}(\text{double}, tx, ty, tz)$$

## Least fix-point:

$$tx = ty = tz = \text{int8}$$

# Simulink type inference

- Fix-point computation on a lattice:
- E.g.:



## Fix-point equations:

$$tx = \text{sup}(\text{double}, tx, ty, tz)$$

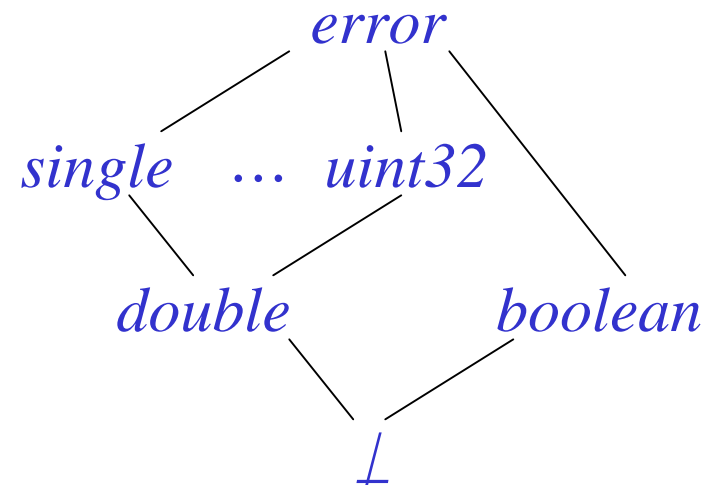
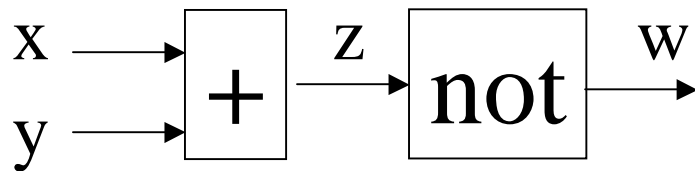
$$ty = \text{sup}(\text{double}, tx, ty, tz)$$

$$tz = \text{sup}(\text{double}, tx, ty, tz, \text{boolean}, tw)$$

$$tw = \text{sup}(\text{boolean}, tz, tw)$$

# Simulink type inference

- Fix-point computation on a lattice:
- E.g.:



Fix-point equations:

$$tx = \text{sup}(\text{double}, tx, ty, tz)$$

$$ty = \text{sup}(\text{double}, tx, ty, tz)$$

$$tz = \text{sup}(\text{double}, tx, ty, tz, \text{boolean}, tw)$$

$$tw = \text{sup}(\text{boolean}, tz, tw)$$

Least fix-point:

$$tx = ty = tz = tw = \text{error}$$

# From Simulink types to Lustre types

- Types are mandatory in Lustre.
- Basic types: *real, int, bool*.
- Complex types, e.g.: *{real, real, int}*.
- Map Simulink types to Lustre types:
  - *double, single*: *real*
  - *int32, int16, int8, ...* : *int*
  - *boolean*: *bool*
- “Mux” translated into complex types.

# Translation steps

1. Type inference
2. Clock inference
3. Bottom-up block-by-block translation

# Time in Lustre

- Lustre clock rule is essentially one:
  - Cannot combine signals of different clocks

$x = 0 \rightarrow \text{pre}(x) + 2;$

$b = \text{true} \rightarrow \text{not pre}(b);$

$y = x \text{ when } b;$

$z = x + y;$  ← **Compiler error**

time	0	1	2	3	4	...
x	0	2	4	6	8	...
b	true	false	true	false	true	...
y	0		4		8	...

# Time in Lustre

- Lustre clock rule is essentially one:
  - Cannot combine signals of different clocks

$x = 0 \rightarrow \text{pre}(x) + 2;$

$b = \text{true} \rightarrow \text{not pre}(b);$

$y = x \text{ when } b;$

$z = x + \text{current}(y);$  ← OK

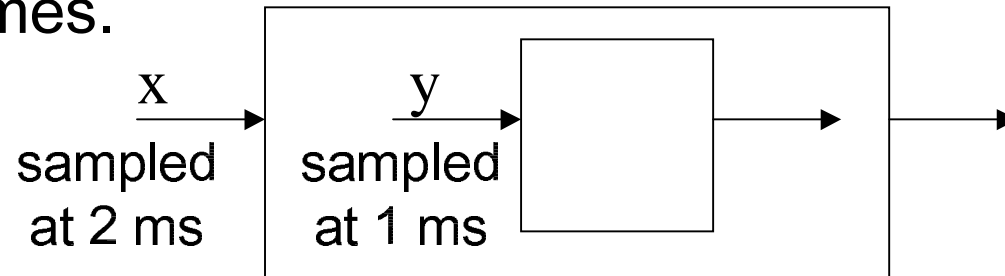
time	0	1	2	3	4	...
x	0	2	4	6	8	...
b	true	false	true	false	true	...
y	0		4		8	...
current(y)	0	0	4	4	8	25

# Time in Simulink

- Simulink has two timing mechanisms:
  - *sample times* : (period,phase)
    - Can be set in blocks: in-ports, UD, ZOH, DTF, ...
    - Defines when output of block is updated.
    - Can also be **inherited** (from inputs or parent system).
  - *triggers* :
    - Set in subsystems
    - Defines when subsystem is “active” (outputs updated).
    - The sample times of all children blocks are inherited.

# Time in Simulink

- **Non-modularity** :
  - A **child** block can go **faster** than the **parent** subsystem: using different (non-inherited) sample times.



- Triggered Simulink systems are modular (very similar to Lustre).

# Time in Simulink

- Timing rules:
  - Can **add up** signals with **different periods** (output will have **GCD** period).
  - When passing from “**slow**” to “**fast**” signals or vice-versa, have to use **special** blocks.
  - Signals should **not feed** blocks with **different periods**.

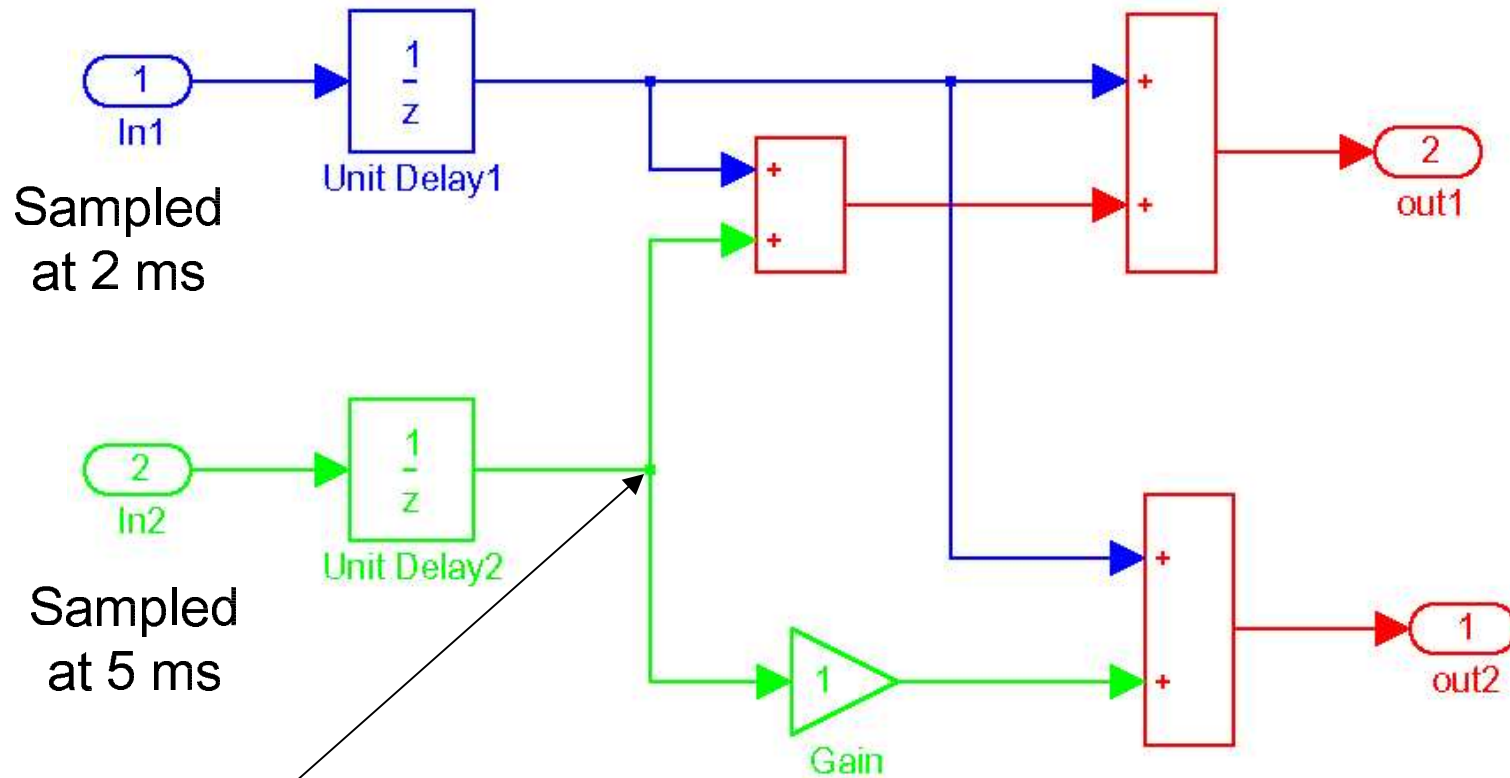
# Overview of clock inference algorithm

1. Infer the sample time of every Simulink signal.
2. Verify Simulink's timing rules.
3. Create Lustre clocks for Simulink sample times.

# Sample time inference

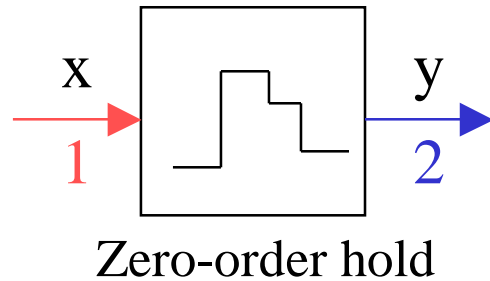
- Basic idea: same as type inference.
- A bit more complicated:
  - Elements of lattice: pairs (period, phase).
  - Lattice is infinite, no *error* element.
  - $p1 \leq p2$  if  $p1$  is “multiple” of  $p2$  (complex def. of multiple because of phase).
  - Sup is “GCD”.

# A model that fails a timing rule



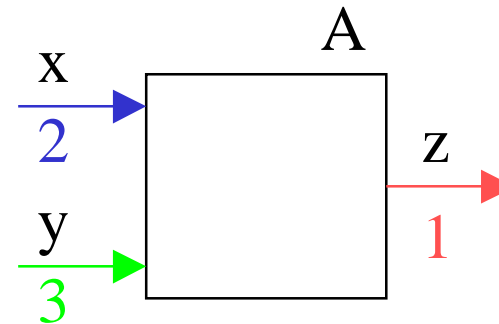
Signal feeds blocks with different periods  
(blue: 2ms, green: 5ms, red: 1ms)

# Associating Lustre clocks



`y = x when basic_1_2;`

`basic_1_2 = T F T F ...`



`xc = current(x);`  
`yc = current(y);`  
`z = A(xc, yc);`

# Translation steps

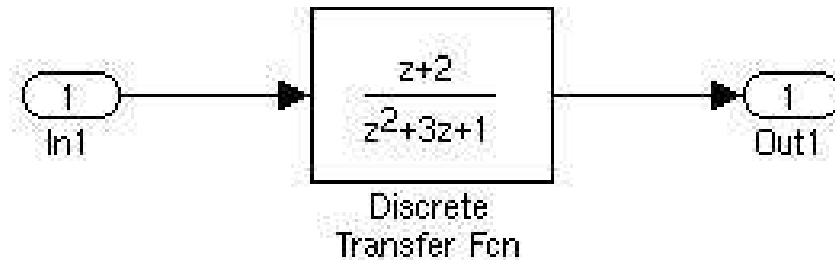
1. Type inference
2. Clock inference
3. Bottom-up block-by-block translation

# Bottom-up translation

- A Simulink model can be seen as a **tree**:
  - root system, subsystems, basic blocks (**leaves**).
- A simple block (+, gain, ...) is translated to a basic Lustre operator (+, \*, ...).
- Complex blocks (transfer functions, ...) are translated into Lustre nodes.
- Subsystems are translated into Lustre nodes.

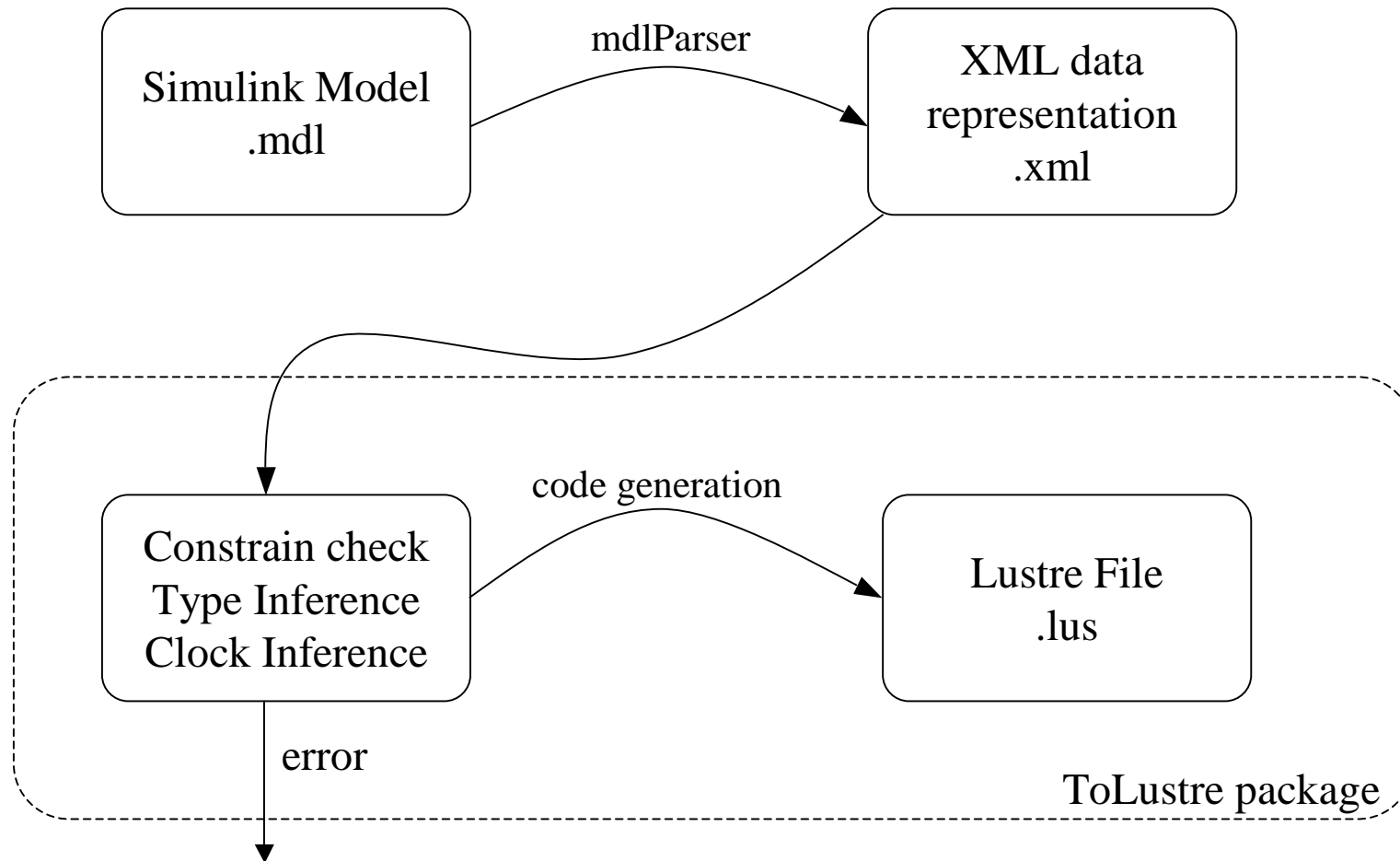
# Bottom-up translation

- Discrete Transfer Function:



```
node Transfer_Function_3(E: real)
returns(S: real);
var Em1, Em2, Sm1, Sm2: real;
let
  S = 1*Em1 + 2*Em2 - 3*Sm1 - 1*Sm2;
  Em1 = 0.0 -> pre(E);
  Em2 = 0.0 -> pre(Em1);
  Sm1 = 0.0 -> pre(S);
  Sm2 = 0.0 -> pre(Sm1);
tel
```

# Prototype tool: Sim2Lus



# Case studies

- Translated two case studies from Audi.
  - A **warning-filtering** system:
    - 6 levels, 20 subsystems, 113 total blocks.
    - 800 lines of generated Lustre code.
  - A **steer-by-wire** application:
    - 6 levels, 18 subsystems, 157 total blocks.
    - 387 lines of generated Lustre code.

# A real-life demo

- Steer-by-wire application provided by Audi.
- Demo planned for January 2004 in Audi's test-track, Ingolstadt, Germany.

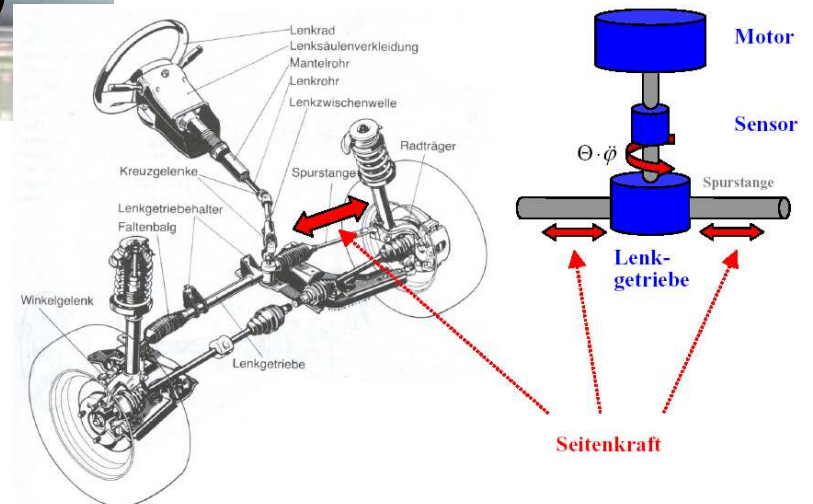


# The Integration platform



## Consists of:

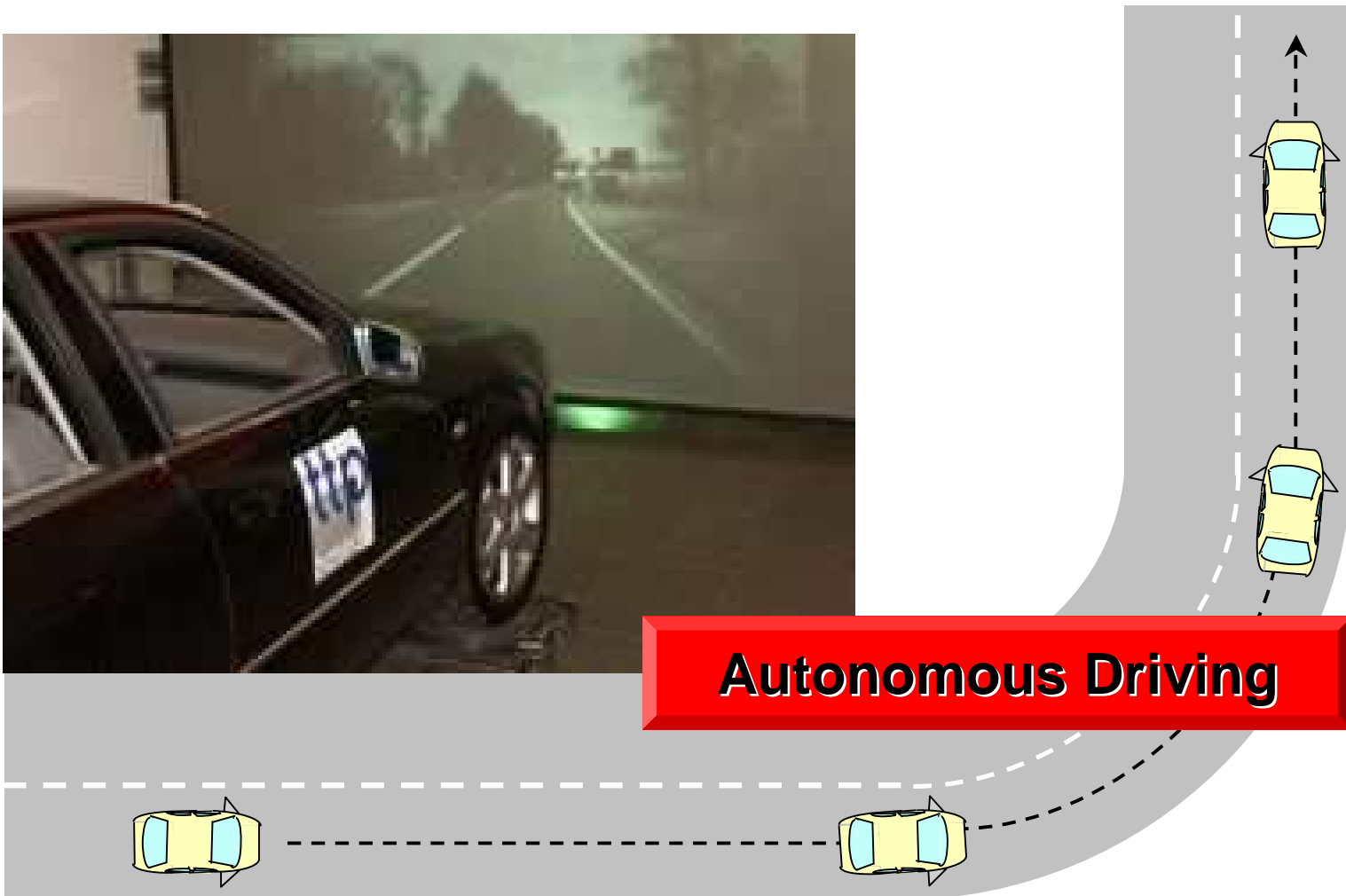
- camera
- steering actuator



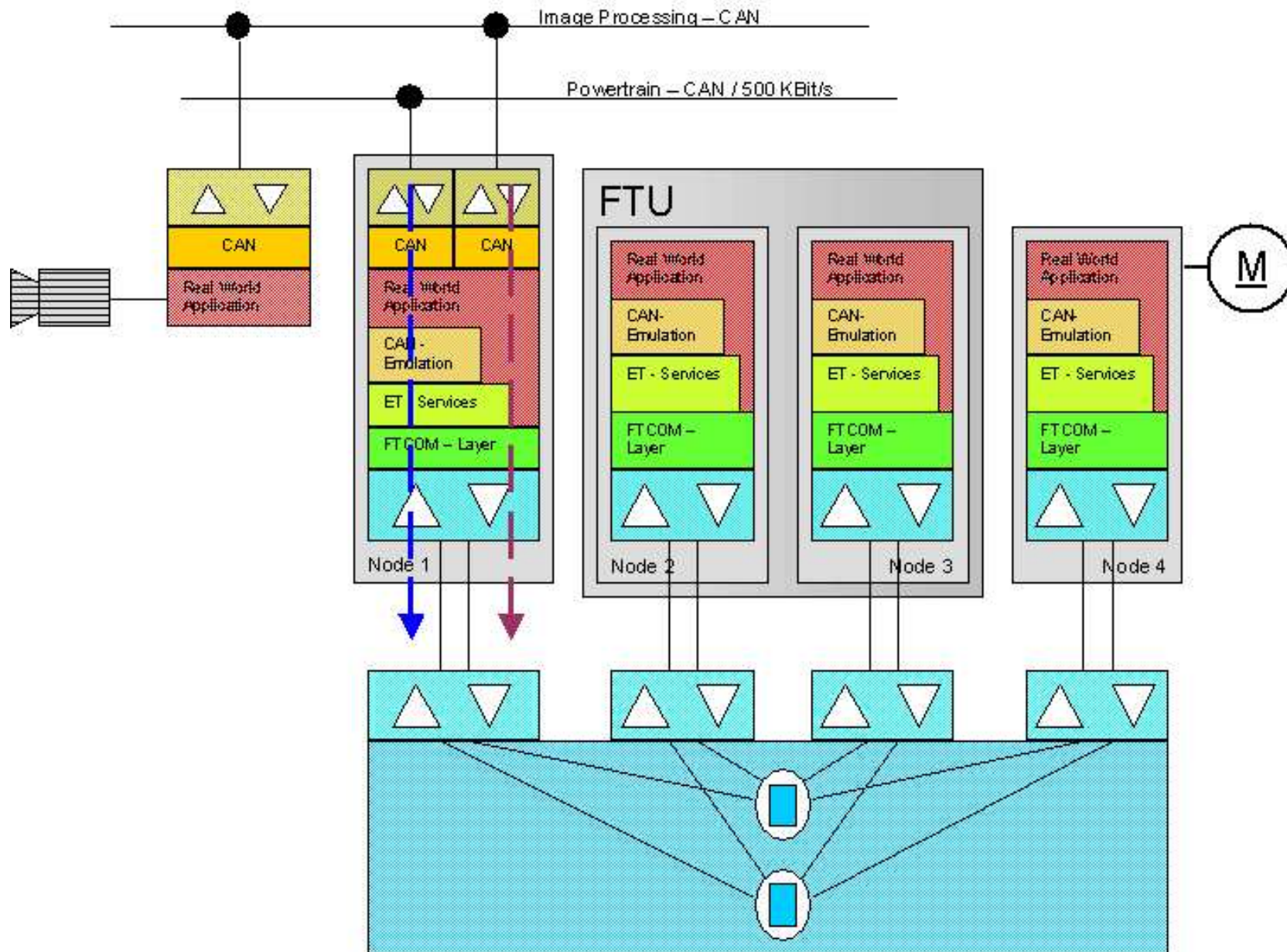
## The Application: steer-by-wire



**Autonomous Driving**



# The steer-by-wire application architecture



# The Simulink model (parts)

