Automatic Code Generation

Several tools allow automatic code generation from high-level control models:

- Simulink Real-Time Workshop (Mathworks)
- Scicos (Inria)
- Lustre/SCADE (Verimag/Esterel-Tecnologies)
- Targetlink (DSpace)
- ASCET (ETAS)

This presentation is mostly based on our experience of the Lustre/SCADE Simulink Gateway and Code Generator.

Model-Based Development

These tools open the way to the Model-Based development method

- Design and validation on models (Simulink, Scicos)
- Automatic deployment on a given architecture guarentying faithfulness

This method avoids the manual coding phase which is error-prone.

It places the computerised control field ahead with respect to other computing fields

Steps in Code Generation

We first focus here on single thread code generation.

- Type inference
- Clock inference
- Code organisation
- Equation sorting
- Optimisation

Type Inference

Simulink/Stateflow is a partially typed system:

The user needs not care about types

Yet the resulting code should be typed.

$$\boxed{\Rightarrow \text{need for type inference}}$$

This is the same situation as in functional languages (OCaml, Haskell). We can apply here the techniques that have been studied there (Hindley-Milner):

- writing type equations
- soving them by unification algorithm

Writing Type Equations

Starting points:

- Blocks have a signature imposing type relations on their inputs and outputs

- The user can impose types in some blocks

Simulink Types

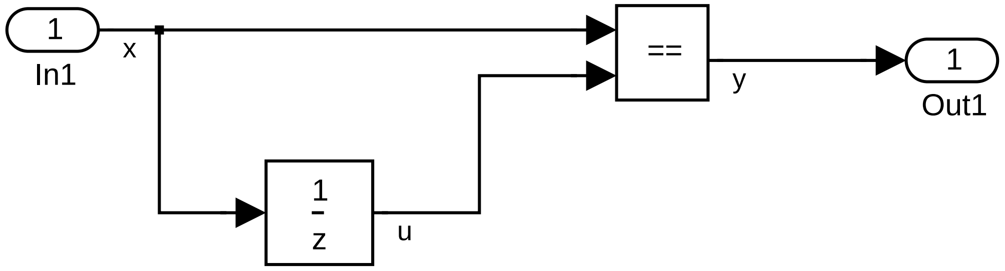
$$\begin{aligned} SimT &= \{type\ variables\} \cup SimNum \cup \{boolean\} \\ SimNum &= \{numerical\ type\ variables\} \cup \\ &\quad \{double,\ single, \\ &\quad int8,\ uint8, \\ &\quad int16,\ unit16, \\ &\quad int32,\ uint32\} \end{aligned}$$

Block Type Signatures

$$\begin{aligned}
\mathit{Constant}_\alpha &: \alpha, \alpha \in SimT \\
\mathit{Adder} &: \alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in SimNum \\
\mathit{Gain} &: \alpha \rightarrow \alpha, \alpha \in SimNum \\
\mathit{Relation} &: \alpha \times \alpha \rightarrow \mathit{boolean}, \alpha \in SimT \\
\mathit{Switch} &: \alpha \times \beta \times \alpha \rightarrow \alpha, \alpha, \beta \in SimNum \\
\mathit{Logical\ Operator} &: \mathit{boolean} \times \cdots \times \mathit{boolean} \rightarrow \mathit{boolean} \\
\mathit{Discrete\ Transfer\ Function} &: \mathit{double} \rightarrow \mathit{double} \\
\mathit{Zero\text{-}Order\ Hold,\ Unit\ Delay} &: \alpha \rightarrow \alpha, \alpha \in SimT \\
\mathit{Data\ Type\ Converter}_\alpha &: \beta \rightarrow \alpha, \alpha, \beta \in SimT \\
\mathit{InPort,\ OutPort} &: \alpha \rightarrow \alpha, \alpha \in SimT
\end{aligned}$$

Writing and Solving Type Equations

Example: the stable subsystem



type equations:

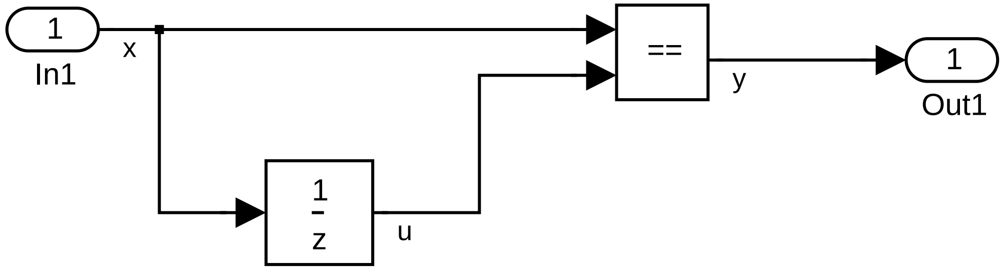
$$\frac{1}{z} \quad : \quad tu = tx$$

$$== \quad : \quad \begin{cases} tx = tu \\ ty = boolean \end{cases}$$

How to solve this system of equations?

Writing and Solving Type Equations

Example: the stable subsystem



type equations:

$$\frac{1}{z} \quad : \quad tu = tx$$

$$== \quad : \quad \begin{cases} tx = tu \\ ty = boolean \end{cases}$$

How to solve this system of equations?

Let's eliminate the type variable tu

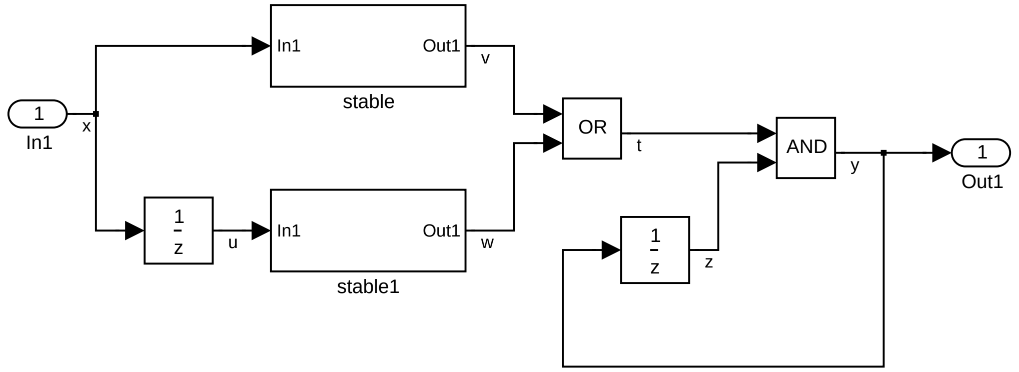
This amounts to unify tx and tx which unify trivially.

Thus we get

$$stable : tx \rightarrow boolean$$

Writing and Solving Type Equations

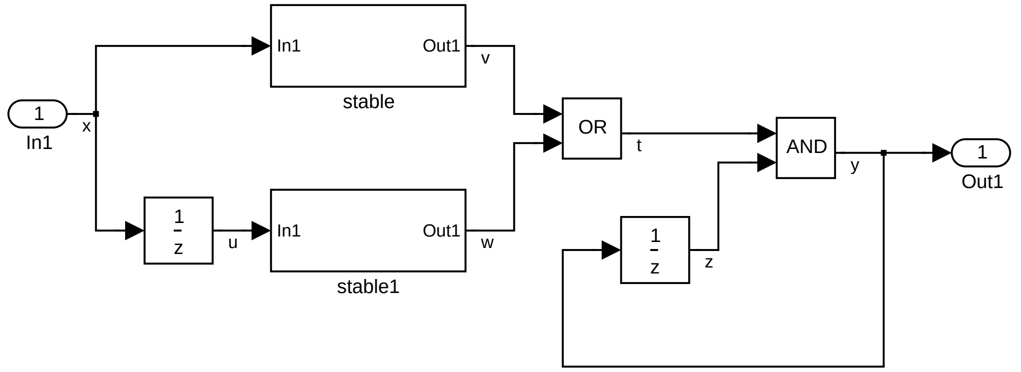
More involved: the monitor subsystem



type equations:

Writing and Solving Type Equations

More involved: the monitor subsystem

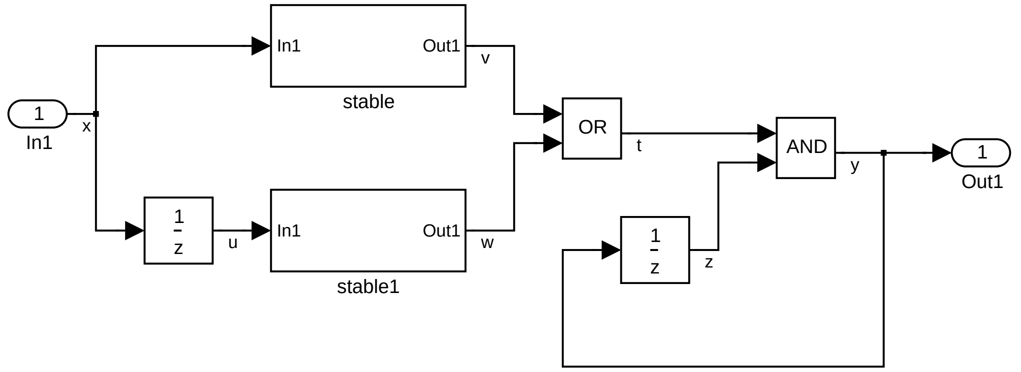


type equations:

$\frac{1}{z}$	:	$tu = tx$
$stable$	:	$tv = boolean$
$stable1$	:	$tw = boolean$
OR	:	$tv = tw = tt = boolean$
AND	:	$tt = tz = ty = boolean$
$\frac{1}{z}$	:	$tz = ty$

Writing and Solving Type Equations

More involved: the monitor subsystem



type equations:

$\frac{1}{z}$	:	$tu = tx$
$stable$	:	$tv = boolean$
$stable1$	:	$tw = boolean$
OR	:	$tv = tw = tt = boolean$
AND	:	$tt = tz = ty = boolean$
$\frac{1}{z}$	:	$tz = ty$

We get $ty = boolean$ and thus:

$$monitor : tx \longrightarrow boolean$$

Other Type Analysis

- Complex Signals
- Signal dimensions

Follow the same lines

Clock Inference

An important question: when should the implementation compute?

A new issue that doesn't exist in classical computing.

Yet the same typing techniques can apply:

- writing clock equations
- soving them by unification

Simulink Clocks

- Periodic clocks defined by:

  a rational positive period $\pi \in Q^+$

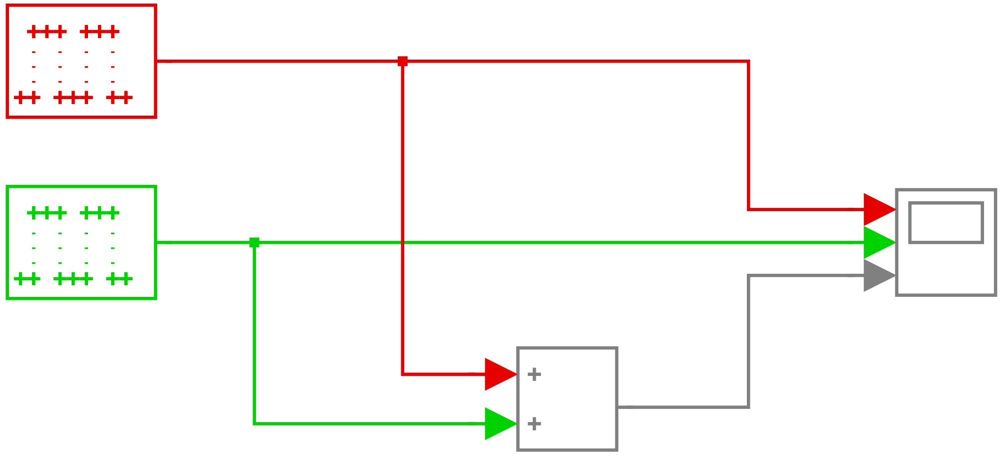
  and a rational positive phase $\theta < \pi$

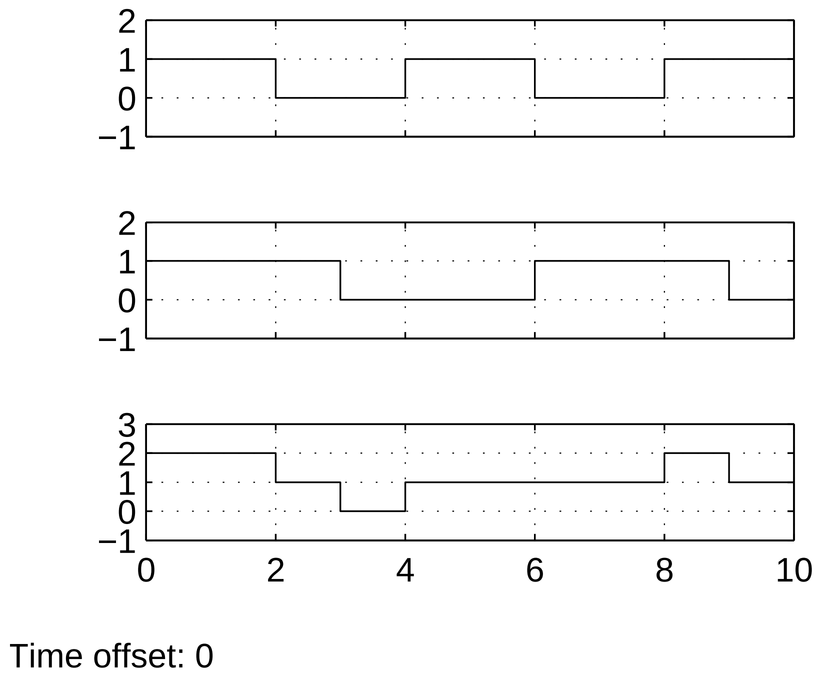
- Triggers, akin to boolean signals

- Clock variables

The GCD Rule

When a block has several inputs with different sample times, the output sample time is given by the generalised gcd rule:

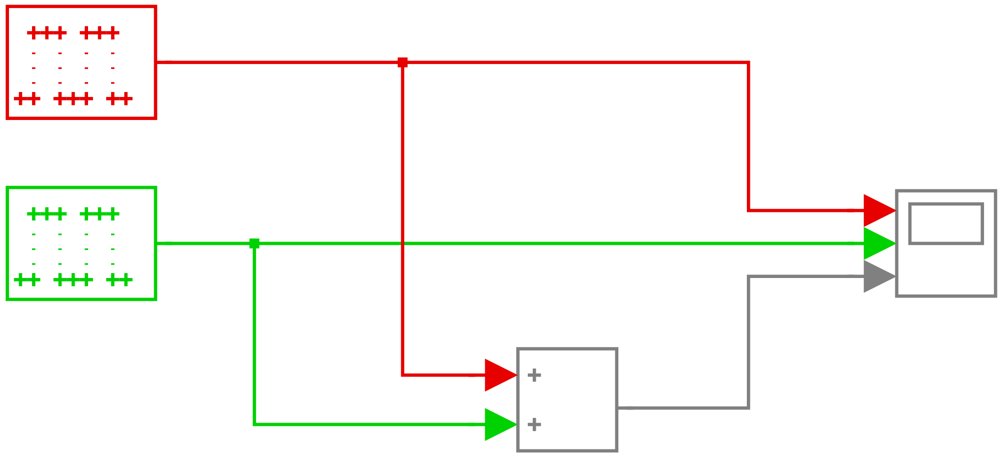


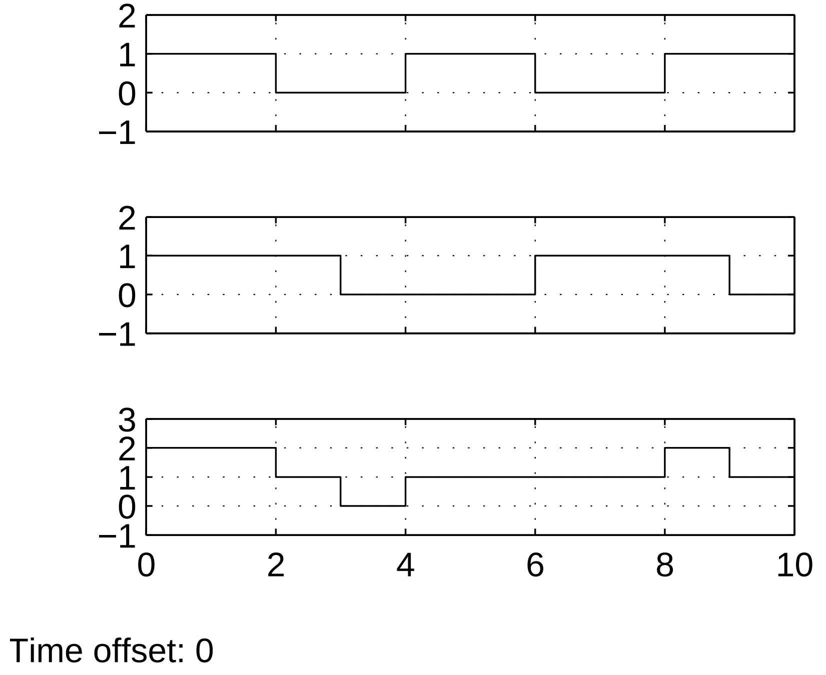


$$ggcd((\pi_1, \theta_1), (\pi_2, \theta_2)) = \begin{cases} (\pi = gcd(\pi_1, \pi_2), \theta_1 \; modulo \; \pi) & if \; \theta_1 = \theta_2 \\ (gcd(\pi_1, \pi_2, \theta_1, \theta_2), 0) & otherwise \end{cases}$$

The GCD Rule

When a block has several inputs with different sample times, the output sample time is given by the generalised gcd rule:





$$\begin{aligned} ggcd((2,0),(3,0)) &= (gcd(2,3),0) \\ &= (1,0) \end{aligned}$$

Useful Simplifications

- $ggcd$ is associative commutative. So we can write:

$$ggcd(a, b, c) = ggcd(a, ggcd(b, c))$$

- $ggcd(a, a, b) = ggcd(a, b)$

- The solution of $a = ggcd(a, b)$

  is $a = b$

Simulink Clock Signatures

$$
\begin{aligned}
\mathit{Sources}_{-1} &: \alpha \\
\mathit{Sources}_{(\pi,\theta)} &: (\pi,\theta) \\
\mathit{Maths}_{-1} &: \alpha_1 \times \alpha_2 \ldots \times \alpha_n \rightarrow ggcd(\alpha_1, \alpha_2, \ldots \alpha_n) \\
\mathit{Maths}_{(\pi,\theta)} &: \alpha_1 \times \alpha_2 \ldots \times \alpha_n \rightarrow (\pi,\theta) \\
\mathit{Discrete}_{-1} &: \alpha \rightarrow \alpha \\
\mathit{Discrete}_{(\pi,\theta)} &: \alpha \rightarrow (\pi,\theta) \\
\mathit{Sinks}_{-1} &: \alpha \rightarrow \alpha \\
\mathit{Sinks}_{(\pi,\theta)} &: \alpha \rightarrow (\pi,\theta)
\end{aligned}
$$

The Triggered Subsystem Inference Rule

$$\begin{array}{llcl} if & S & : & \alpha^m \rightarrow \alpha^n \\ and & b & : & \beta \\ \hline then & S^{\hat{b}} & : & b^m \rightarrow \beta^n \end{array}$$

where $S^{\hat{b}}$ means: the subsystem S triggered by the signal b

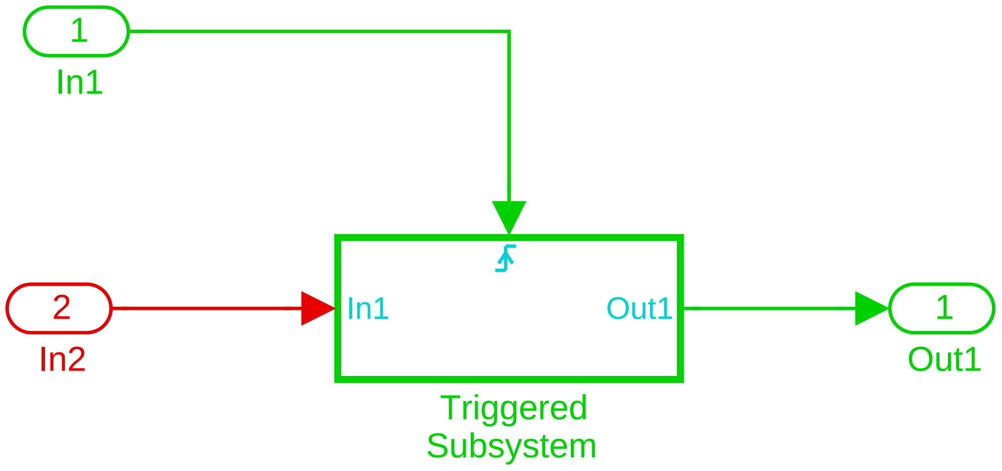
This amounts to saying that a triggered subsystem should have a single clock and that once triggered by a signal, it executes only when the trigger is active and then its outputs are hold when the trigger is inactive.

The Triggered Subsystem Inference Rule

$$\begin{array}{llll} if & S & : & \alpha^m \rightarrow \alpha^n \\ and & b & : & \beta \\ \hline then & S^{\hat{b}} & : & b^m \rightarrow \beta^n \end{array}$$

where $S^{\hat{b}}$ means: the subsystem S triggered by the signal b

This amounts to saying that a triggered subsystem should have a single clock and that once triggered by a signal, it executes only when the trigger is active and then its outputs are hold when the trigger is inactive.



Clocking the single thread case

Clock inference provides us with a basic clock

- either the fastest periodic clock
- or a clock variable, if there is no periodic clock in the design

In any case, every other clock of the system is slower and can be considered as a trigger.

In case of periodic clocks, these triggers are implicite and we must make them explicit by means of clock dividers and phasers

Clocking the single thread case

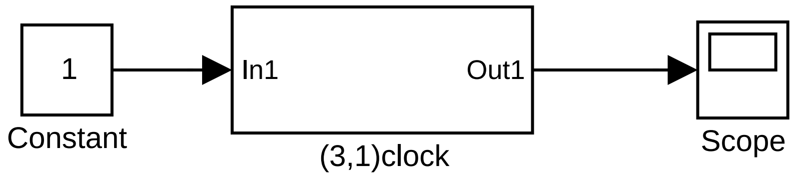
Clock inference provides us with a basic clock

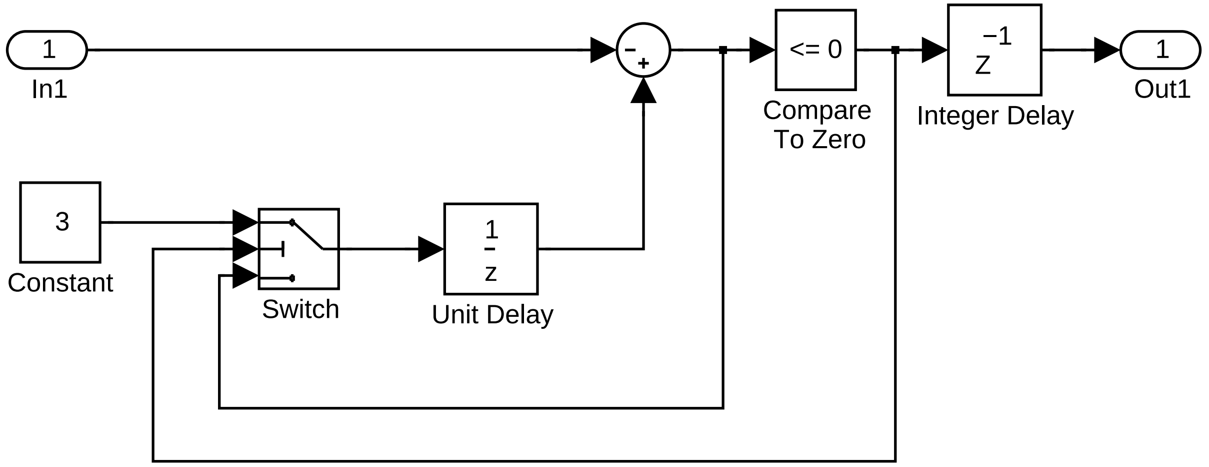
- either the fastest periodic clock
- or a clock variable, if there is no periodic clock in the design

In any case, every other clock of the system is slower and can be considered as a trigger.

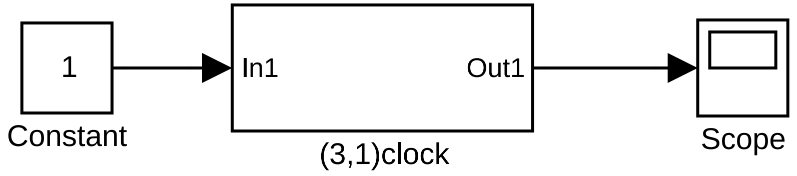
In case of periodic clocks, these triggers are implicit and we must make them explicit by means of clock dividers and phasers

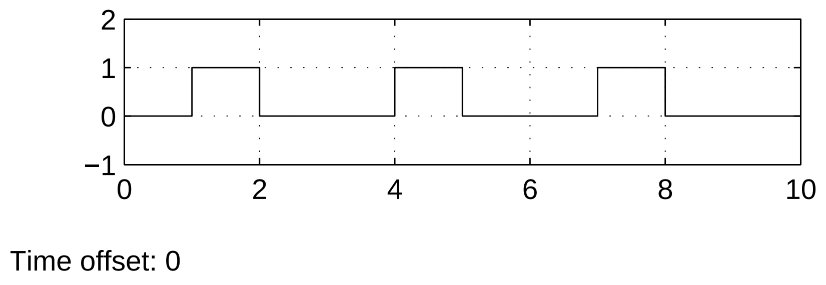
Example: A (3,1) clock

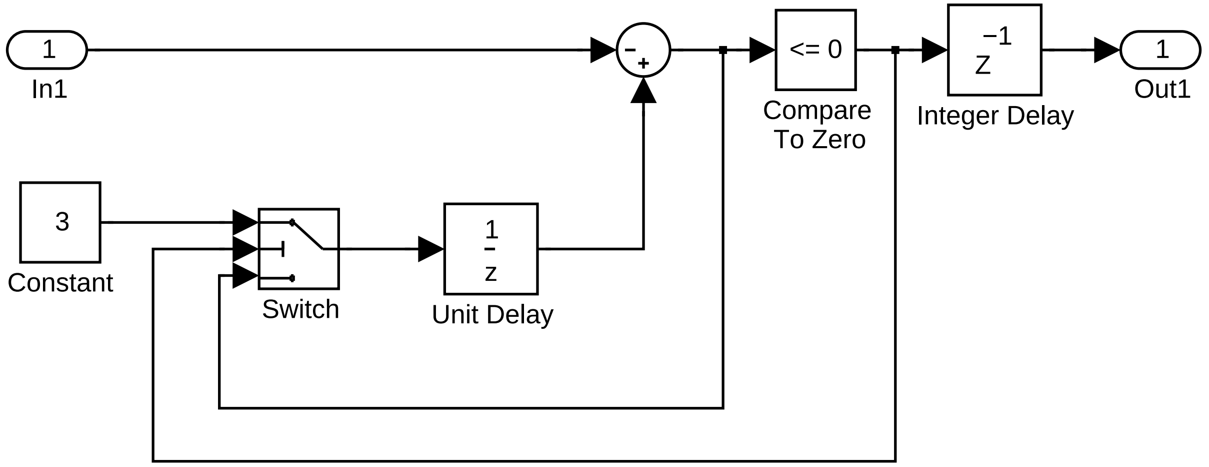




Example: A (3,1) clock







From Subsystems to Difference Equations

A single-clock subsystem is a system of difference equations

$$\begin{array}{lll} U(0), & & \\ U(n+1) & = & F(U(n), X(n)) \\ Y(n) & = & G(U(n), X(n)) \end{array}$$

where

- U, vector of state variables
- X, vector of inputs
- Y, vector of outputs
- F, state transition function,
- G, output function,

From Difference Equations to Programs

The idea is to build on object-oriented programming (C^{++}, Java)

Simulink	Object-Oriented
system of equations	class
state variables	class attributes
initialisation	object creation
functions	class methods

From Difference Equations to Programs

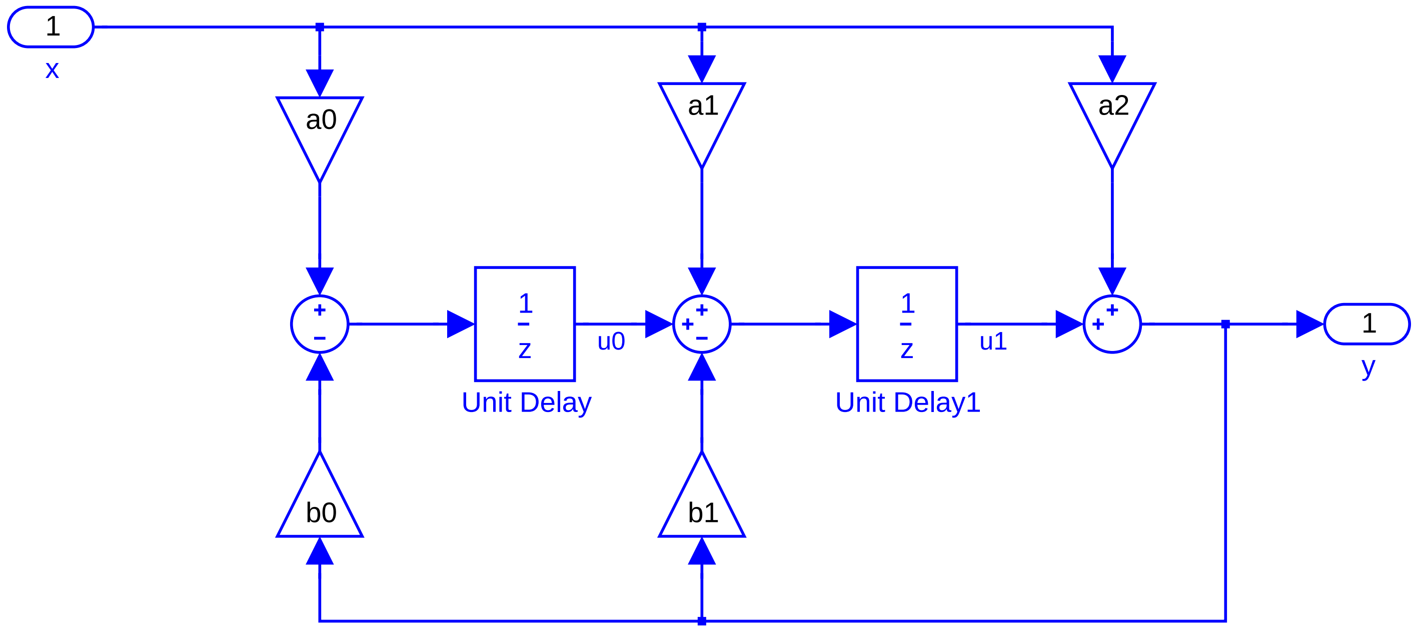
$U(0)$,

$$\begin{aligned} Y(n) &= G(U(n), X(n)) \\ U(n+1) &= F(U(n), X(n)) \end{aligned}$$

```
class subsyst{
private :
  state u ;
public :
  subsyst(state u_init){
    u = u_init;
  }
  output step(input x){
    state up;
    output y = g(u, x);
    up = f(u, x);
    u = up;
    return y;
  }
}
```

A local version of state variables is used so as to not depend on the order of computations.

Example



$$
\begin{aligned}
U_0(z) &= z^{-1}(a_0X(z) - b_0Y(z)) \\
U_1(z) &= z^{-1}(a_1X(z) - b_1Y(z) + U_0(z)) \\
Y(z) &= a_2X(z) + U_1(z)
\end{aligned}
$$

From Simulink to Difference Equations

$$\begin{aligned} U_0(z) &= z^{-1}(a_0X(z) - b_0Y(z)) \\ U_1(z) &= z^{-1}(a_1X(z) - b_1Y(z) + U_0(z)) \\ Y(z) &= a_2X(z) + U_1(z) \end{aligned}$$

$$\begin{aligned} U_0(n+1) &= a_0X(n) - b_0Y(n) \\ U_1(n+1) &= a_1X(n) - b_1Y(n) + U_0(n) \\ Y(n) &= a_2X(n) + U_1(n) \end{aligned}$$

From Difference Equations to Programs

$$\begin{aligned}
Y(n) &= a_2 X(n) + U_1(n) \\
U_0(n+1) &= a_0 X(n) - b_0 Y(n) \\
U_1(n+1) &= a_1 X(n) - b_1 Y(n) + U_0(n)
\end{aligned}$$

```
class second_order {
  private :
    double a0, a1, a2, b0, b1;
    double u0, u1 ;
 public :
   second_order(double u0, u1)
     {....}
   double step(double x) {
     ...
     return y;
  }
```

Building the step function

$$\begin{aligned} U_0(n+1) &= a_0X(n) - b_0Y(n) \\ U_1(n+1) &= a_1X(n) - b_1Y(n) + U_0(n) \\ Y(n) &= a_2X(n) + U_1(n) \end{aligned}$$

```
double step(double x) {
  double y, up0, up1;
  y   = a2*x + u1 ;
  up0 = a0*x - b0*y ;
  up1 = a1*x - b1*y + u0;
  u0  = up0;
  u1  = up1;
  return y;
}
```

Optimisations

Many optimisations are possible, due to the equational semantics of block-diagrams

- elimination of unused buffers
- eliminationg buffers by reordering computations
- mofify the original system by z^{-1} comutation

Unused Buffers

```
double step(double x) {
  double y, up0, up1;
  y   = a2*x + u1 ;
  up0 = a0*x - b0*y ;
  up1 = a1*x - b1*y + u0;
  u0  = up0;
  u1  = up1;
  return y;
  }
```

```
double step(double x) {
  double y, up0;
  y   = a2*x + u1 ;
  up0 = a0*x - b0*y ;
  u1  = a1*x - b1*y + u0;
  u0  = up0;
  return y;
  }
```

`u1` is not used since `up1` is computed

Reordering Computations

```
double step(double x) {
  double y, up0;
  y    = a2*x + u1 ;
  up0  = a0*x - b0*y ;
  u1   = a1*x - b1*y + u0;
  u0   = up0 ;
  return y;
  }
```

```
double step(double x) {
  double y;
  y    = a2*x + u1 ;
  u1   = a1*x - b1*y + u0;
  u0   = a0*x - b0*y ;
  return y;
  }
```

`u1` depends on `u0` but not the converse. We can compute `u1` first.

Commutating Delays

This is due to the property:

if f a static function (not encompassing z operators), then

$$zf(x, y) = f(zx, zy)$$

Note that f can be non-linear, boolean,...

Known as “Leiserson & Saxe retiming”

Used in hardware generation, pipelining,…

Is this obvious ???

Optimisations

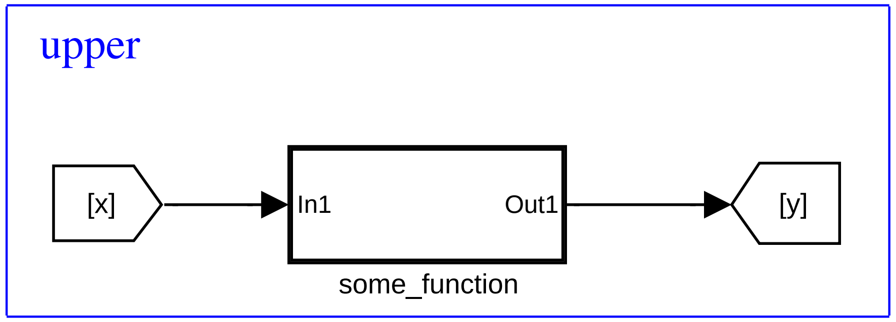
Many possibilities

But the problem of buffer optimisation is a “hard problem”

Heuristics are required and obtaining “good code” is not obvious

Yet preferable to human coding

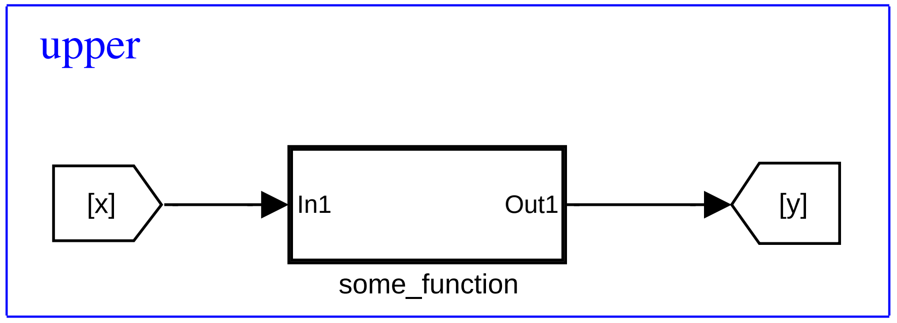
Subsystems



```
class upper{
  private:
    ...
  public:
    output step(inputs...){
      ...
      y = some_function.step(x);
      ...
    }
}
```

Code generation is modular

Subsystems

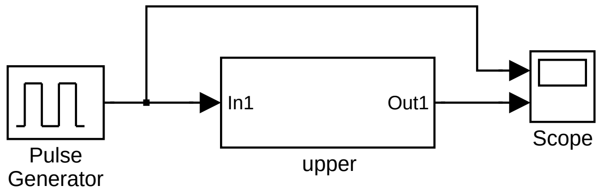


```
class upper{
  private:
    ...
  public:
    output step(inputs...){
      ...
      y = some_function.step(x);
      ...
    }
}
```

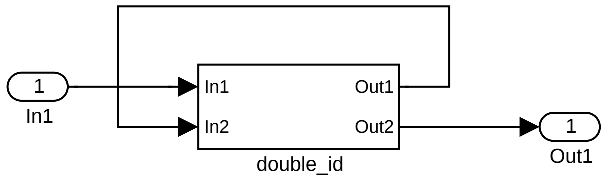
Code generation is modular

But for false causality loops!

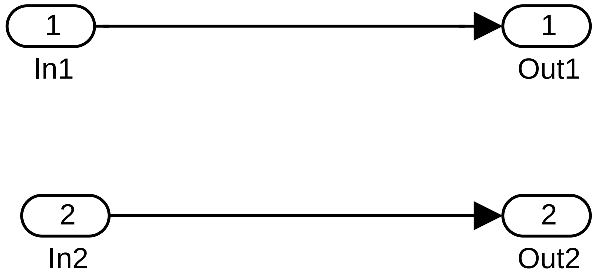
False Causality Loops



upper



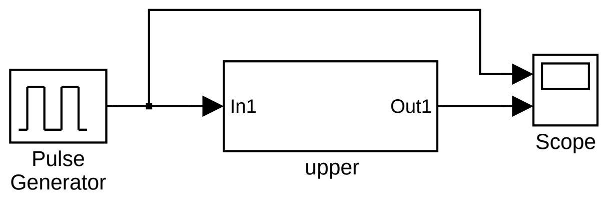
double_id

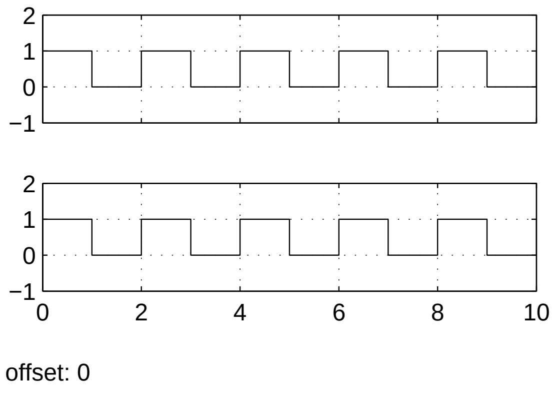


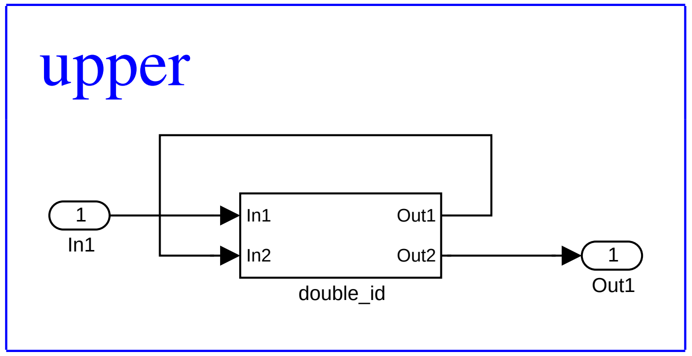
Is this a correct model ?

What is the result ?

False Causality Loops



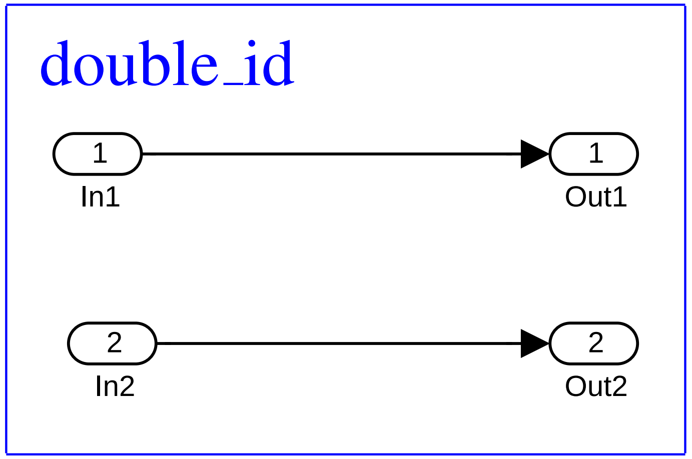




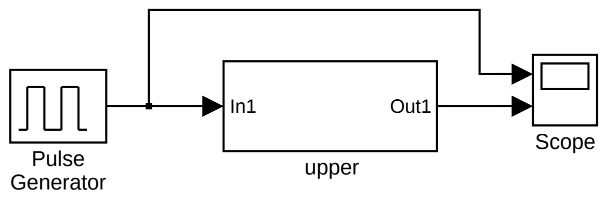
There is no causality loop

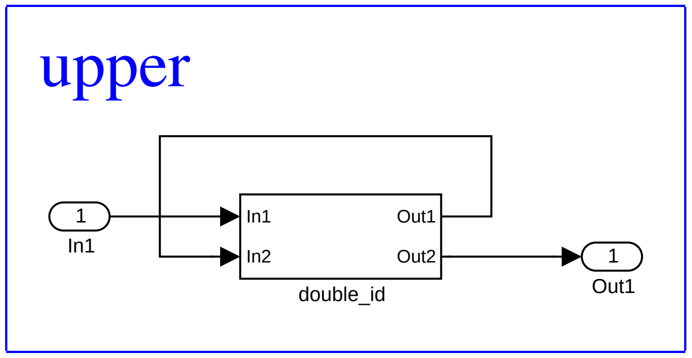
But modular code generation

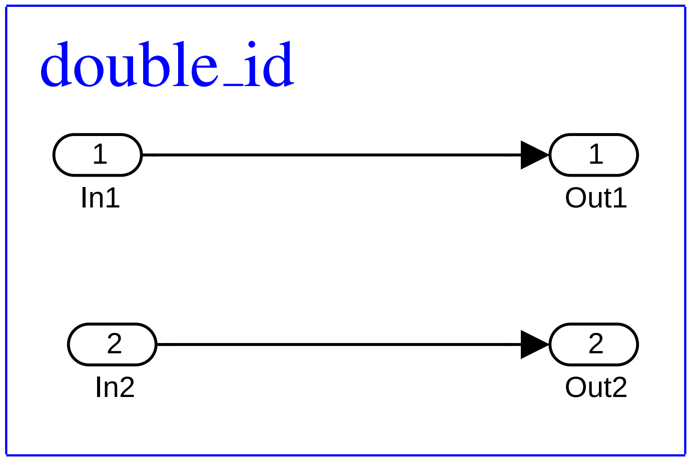
is not possible

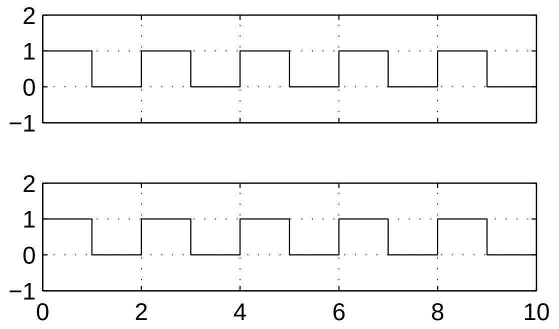


False Causality Loops









Time offset: 0

There is no causality loop

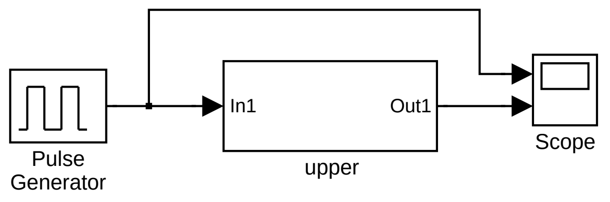
But modular code generation

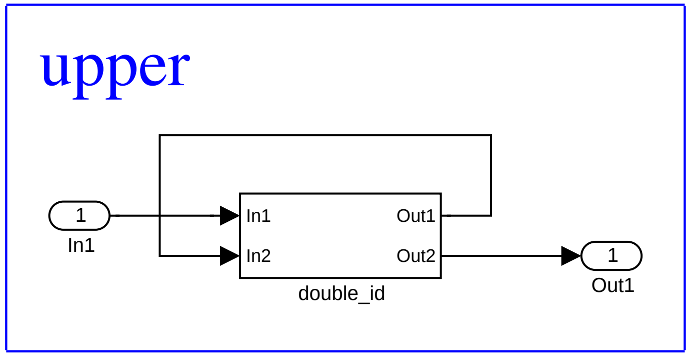
is not possible

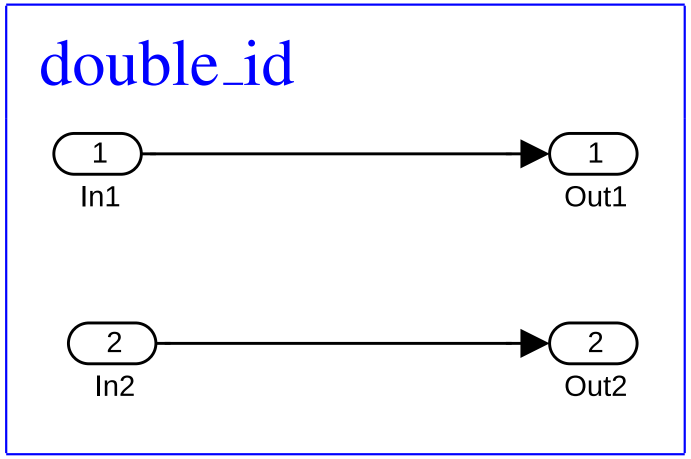
However code can be generated

by inlining

False Causality Loops







Inlining

```
class upper{
public:
  upper(){}
  a_type step(a_type In1){
    a_type double_id_In1 = In1;
    a_type double_id_Out1 = double_id_In1;
    a_type double_id_In2 = double_id_Out1;
    a_type double_id_Out2 = double_id_In2;
    a_type Out1 = double_id_Out2;
    return Out1;
  }
}
```

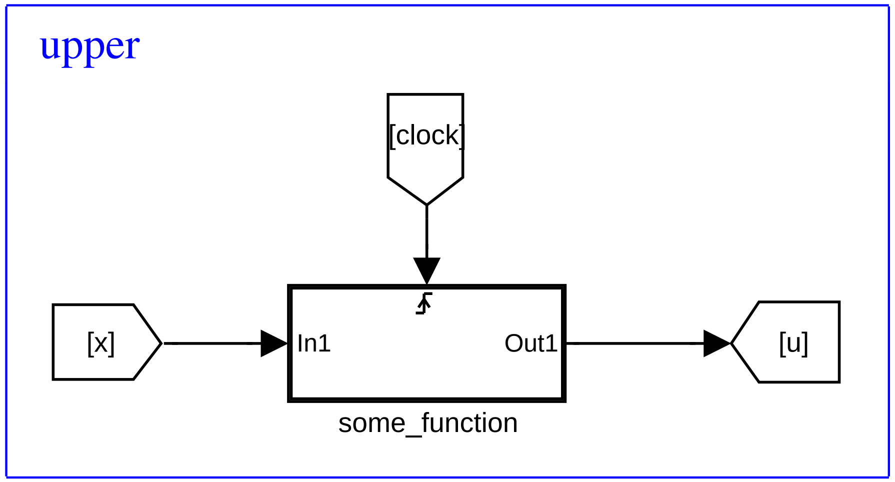
Inlining vs Modularity

Inlining allow generating code for more models

Yet modular code generation is often preferred:

- Better readability
- Easier testing

Triggered Subsystems



```
class upper{
  private:
    ...
    some_type u=0;
    ...

  public:

  output step(inputs...){
    ...
    if (clock)
       u = some_function.step(x);
    ...
  }
}
```

The outputs of triggered subsystems are *state variables* that have to be initialised properly.

Code Generation for Verification

Verification of Simulink/Stateflow through SCADE

Uses the translation to Lustre

- Through the Prover Plugin:

  Translation to TECLA the internal language of the Prover Engine

  Then proof by induction using the Prover SAT - solver.

- using Lustre tools

  for instance, enumerative model-checking through autmata code generation.

Automaton Code Generation

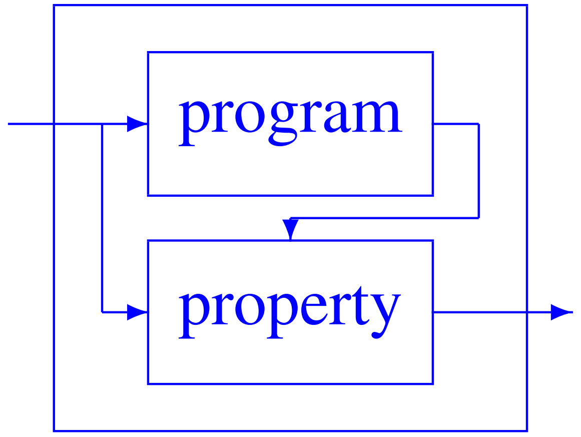
A boolean Lustre program:

- $U(n+1) = F(U(n), X(n))$: State
- $Y(n) \ = G(U(n), X(n))$: Output
- $U(0)$: initial state

Enumerate all the reachable states and for each reachable state and possible input value, the corresponding output value.

Application to code generation

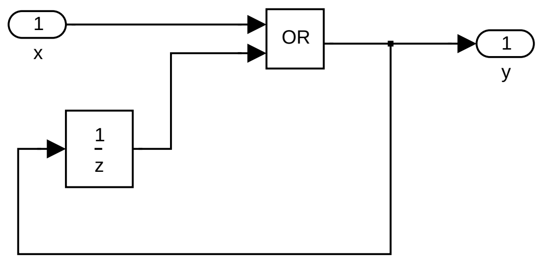
Build the “verification program”:



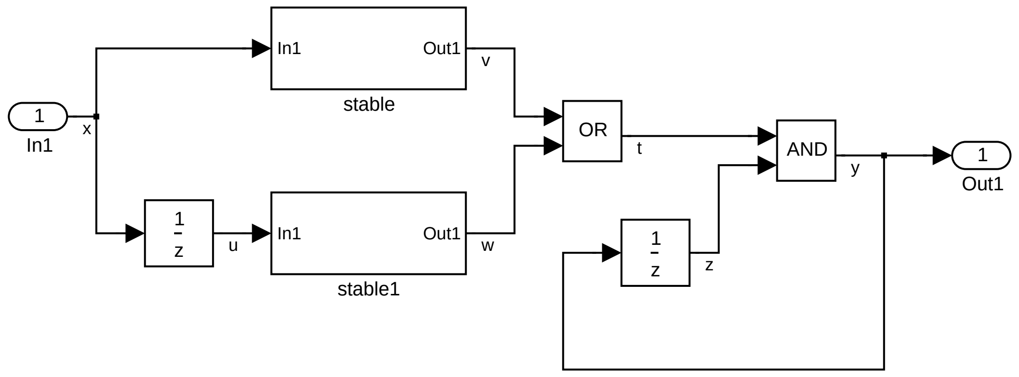
If the program is purely boolean and satisfies the property, the code generation into automaton should never output the value “false”

Example

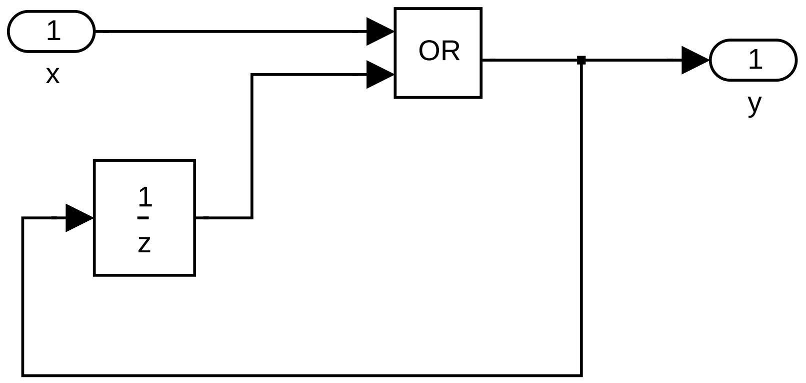
Does this program:



satisfies this property

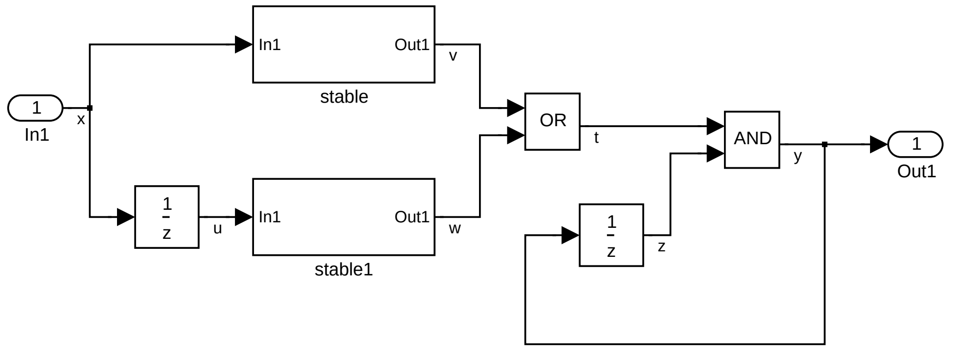


Translation to Lustre



```
node once(x: bool) returns (y: bool);
let  y = x or (false -> pre y);
tel
```

Translation to Lustre



```
node monitor(x: bool) returns(prop: bool);
let
prop = (stable(x) or stable(false -> pre x))
       and (true -> pre prop);
tel
```

Verification

We build the verification node

```
node verif(x: bool) returns (prop : bool);
var y: bool;

let  prop = monitor(y);
     y = once (x);
tel
```

and we generate the automaton

Result

```
   switch(ctx->current_state){
   case 0:
      ctx->_V2 = _true;
      verif_O_prop(ctx->client_data, ctx->_V2);
      if(ctx->_V1){
         ctx->current_state = 1; break;
      } else {
         ctx->current_state = 2; break;
      }
   case 1:
      ctx->_V2 = _true;
      verif_O_prop(ctx->client_data, ctx->_V2);
      ctx->current_state = 3; break;
   case 2:
      ctx->_V2 = _true;
      verif_O_prop(ctx->client_data, ctx->_V2);
      if(ctx->_V1){
         ctx->current_state = 1; break;
      } else {
         ctx->current_state = 2; break;
      }
   case 3:
      ctx->_V2 = _true;
      verif_O_prop(ctx->client_data, ctx->_V2);
      ctx->current_state = 3; break;
   }
}
```

Conclusions on Single-Thread Code Generation

It allows generating code for any discrete-time model that can be simulated.

Allows many optimisations

The need for Real-Time Operating System is minimised

Provides in general robust and efficient code