The Signal synchronous language: the principles beyond the language and how to exploit and extend them

Albert Benveniste and Thierry Gautier (Inria-Rennes)

Acknowledgement: Paul Le Guernic and Loïc Besnard

Collège de France, Mars 2018

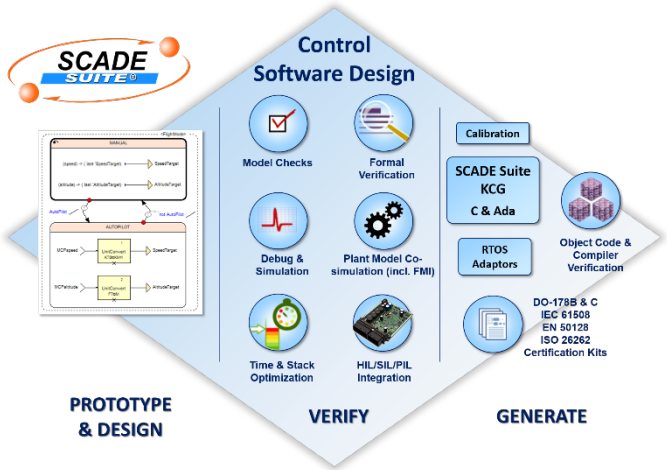
Signal: an original positioning in the landscape of synchronous languages

Albert Benveniste and Thierry Gautier

Mars 2018

Lustre
dataflow functional languages

Lustre, Lucid Synchrone, Scade, (Zélus)



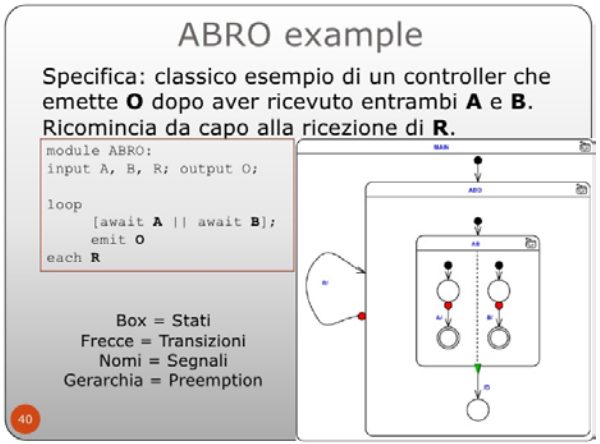
- Streams (seq. of values)
- Dataflow composition à la Kahn: functional

- Simple
- No delay-free loop
- Higher order: dynamicity

- (Clocks as types)

Esterel
imperative languages

Esterel, SyncCharts, SCL/SCCharts, ReactiveML, the web



- variables and values, await, emit, ||, preemption
- Difficulty: combining || and immediate control passing
- Reaction as a fixpoint problem: 0/**1**/several solutions

Signal
equation based language

Open systems and architecture modeling:

- Synchronization
- Clocks as 1st class citizens

A program can have 1000's of clocks ⇒ clocks must be synthesized, not verified

- (clocks as types in Lustre ⇒ "condact" used in Scade)

- Clock equations + Dataflow expressions
- Nondeterminism (but controlled)
- Open systems: stuttering invariance

  (a system has always the provision to sleep while its environment acts)
- Difficulty: Clocks ↔ Data

Contents

1. Signal in the landscape of synchronous languages
2. The Signal vintage watch
3. The clock and causality calculus
4. Zooming on the causality calculus
5. Zooming on the clock calculus
6. Beyond the clock calculus
7. Beyond the causality calculus: upgrading Signal to support data constraints

The Signal vintage watch

Albert Benveniste and Thierry Gautier

An example of Signal program and its compilation

Intuitive pseudo-code


```
X := pre(X)-1
     reset IN every pre(X)≤0
```

Input **IN** returns **X** (mmmmhhh??)

IN is provided only when used

An example of Signal program and its compilation

```
(  X := IN default ZX-1     stream funct
| ZX := X$1 init 0          stream funct
| IN ^= when (ZX ≤ 0) )     clock eqn
```

Signal code

Intuitive pseudo-code

```
X := pre(X)-1
     reset IN every pre(X)≤0
```

Input **IN** returns **X** (mmmmhhh??)

IN is provided only when used

An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream funct
| ZX := X$1 init 0           stream funct
| IN ^= when (ZX ≤ 0) )      clock eqn
```

IN	2			3				5	
ZX	0	2	1	0	3	2	1	0	5
X	2	1	0	3	2	1	0	5	4

Input **IN** returns **X** (mmmmhhh??)
IN is provided only when used

An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream funct
| ZX := X$1 init 0           stream funct
| IN ^= when (ZX ≤ 0) )      clock eqn
```

IN	2			3				5	
ZX	0	2	1	0	3	2	1	0	5
X	2	1	0	3	2	1	0	5	4

IN is schizophrenic: its value is an input of the program but its clock (instants of presence) is not

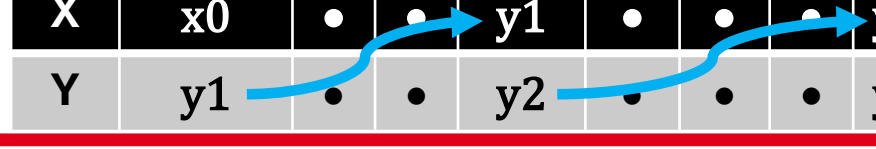
Signal

```
X := f(U,V)
```

X	f(u,v)	•	•	•	f(u,v)	•	•	•	
U	u1	•	•	•	u2	•	•	•	
V	v1	•	•	•	v2	•	•	•	

```
X := Y$1 init X0
```

X	x0	•	•	y1	•	•	•	y2	
Y	y1	•	•	y2	•	•	•	y3	



•: absence (stuttering invariance)

Signal

`X := f(U,V)`

X	f(u,v)	•	•	•	f(u,v)	•	•	•	
U	u1	•	•	•	u2	•	•	•	
V	v1	•	•	•	v2	•	•	•	

`X := Y$1 init X0`

X	x0	•	•	y1	•	•	•	y2	
Y	y1	•	•	y2	•	•	•	y3	

`X := U default V`

X	u1	•	•	•	v2	•	u2	•	
U	u1	•	•	•	•	•	u2	•	
V	v1	•	•	•	v2	•	•	•	

`X := Y when B`

X	y	•	•	•	yk	•	•	•	
Y	y1				yk				
B	True				True				

`X := f(U,V)`

X	f(u,v)	•	•	•	f(u,v)	•	•	•	
U	u1	•	•	•	u2	•	•	•	
V	v1	•	•	•	v2	•	•	•	

`X := Y$1 init X0`

X	x0	•	•	y1	•	•	•	y2	
Y	y1	•	•	y2	•	•	•	y3	

`X := U default V`

X	u1	•	•	•	v2	•	u2	•	
U	u1	•	•	•	•	•	u2	•	
V	v1	•	•	•	v2	•	•	•	

`X := Y when B`

X	y	•	•	•	yk	•	•	•	
Y	y1				yk				
B	True				True				

`K ^= H` equality of clocks: a constraint

An example of Signal program and its compilation

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= (when B)
|  H ^= B ^= X ^= ZX )
```

stream func
stream func
stream func
clock eqn
clock eqn

```
[B]: when B
```

Expanded as

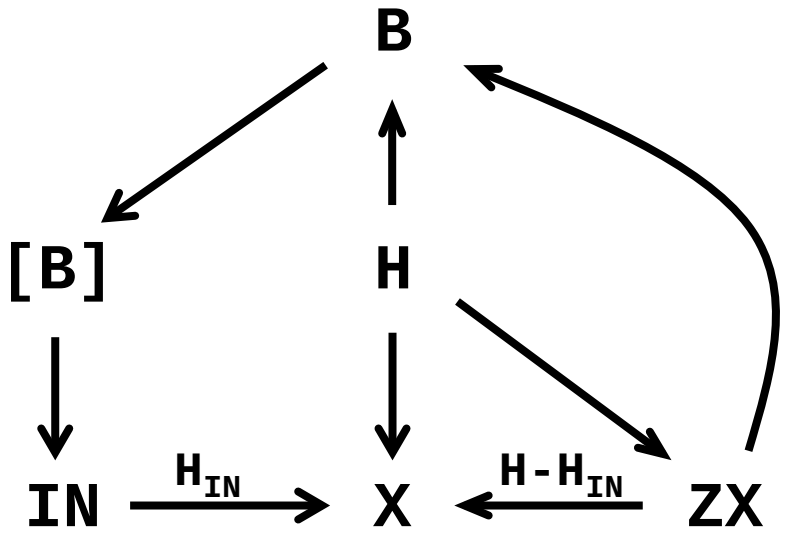
```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )       clock eqn
```

[B]: when B

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

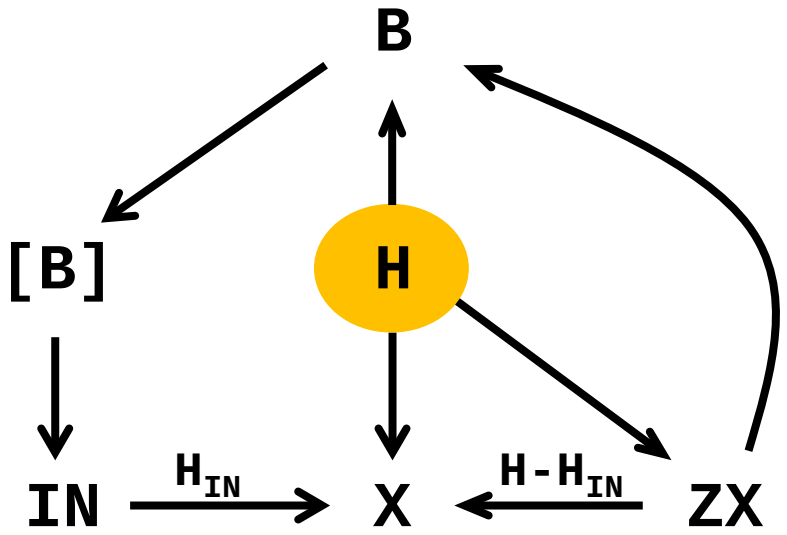


An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )       clock eqn
```

[B]: when B

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

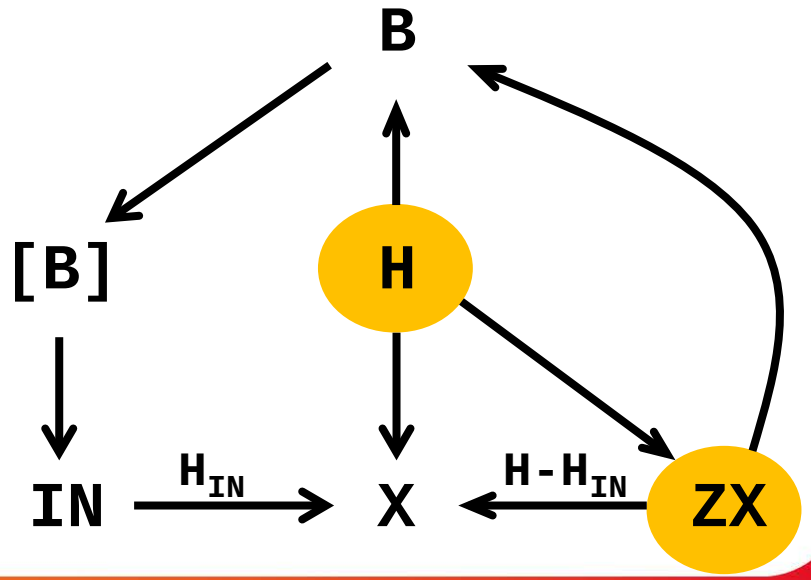


An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )       clock eqn
```

[B]: when B

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

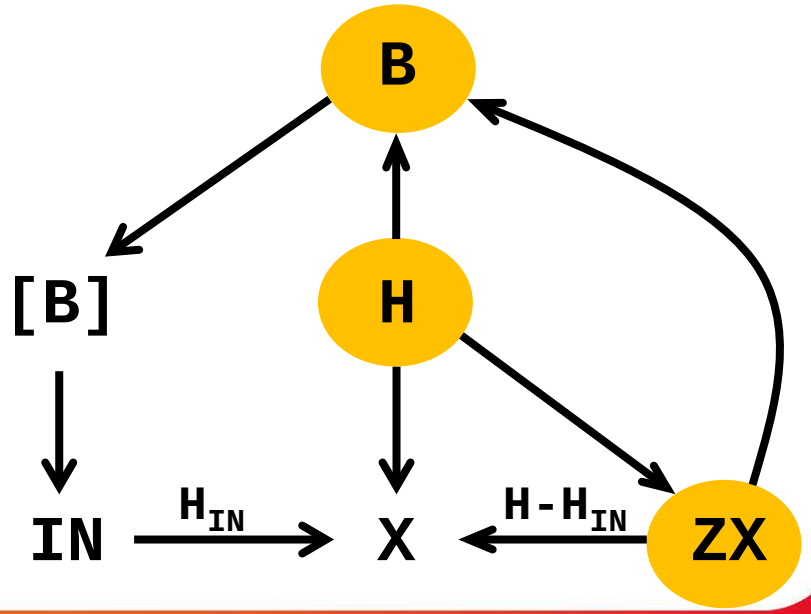


An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )       clock eqn
```

[B]: when B

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

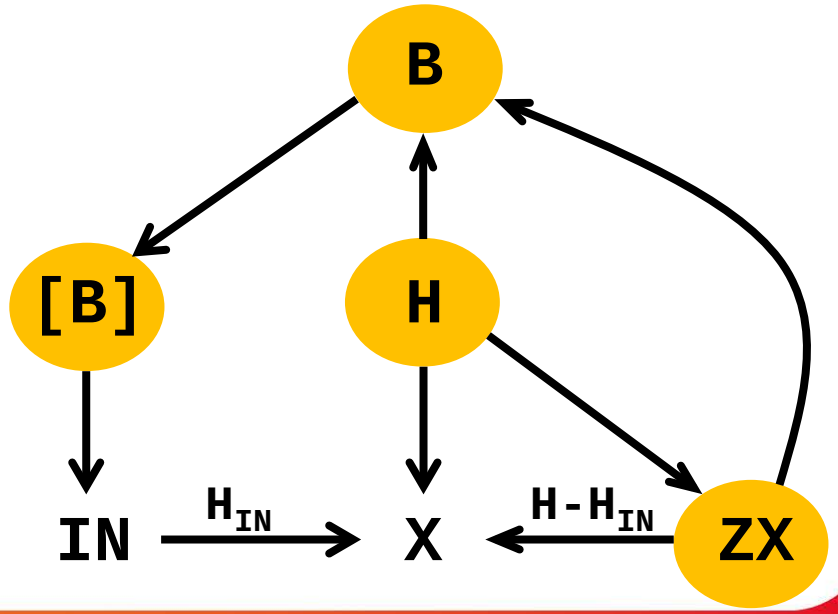


An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )       clock eqn
```

[B]: when B

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

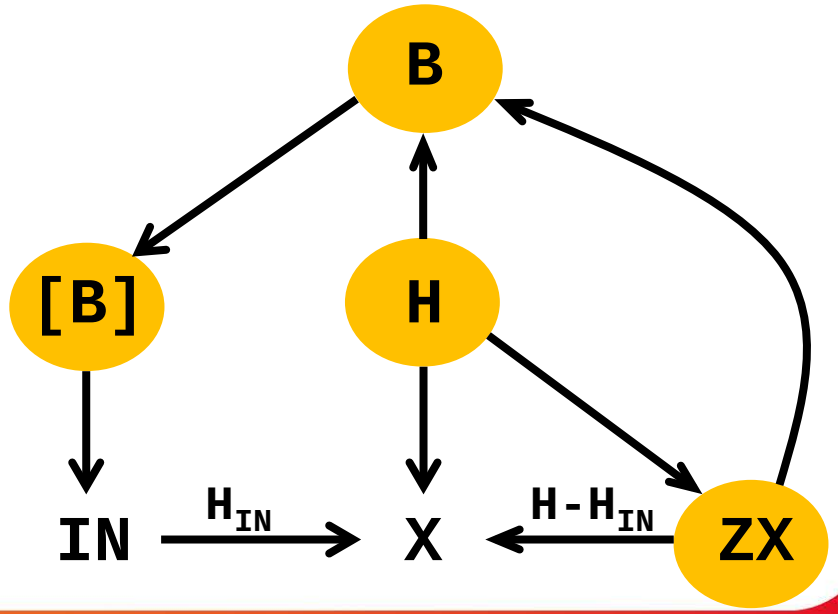


An example of Signal program and its compilation

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= (when B)
|  H ^= B ^= X ^= ZX )
```

stream func
stream func
stream func
clock eqn
clock eqn

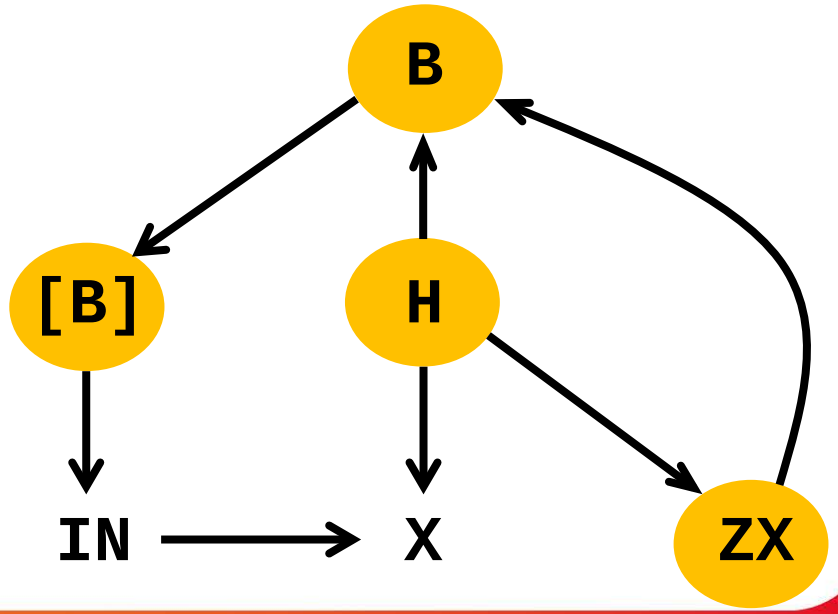
```
[B]: when B
case B true
case B false
```



An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )        clock eqn
```

```
[B]: when B
case B true
```

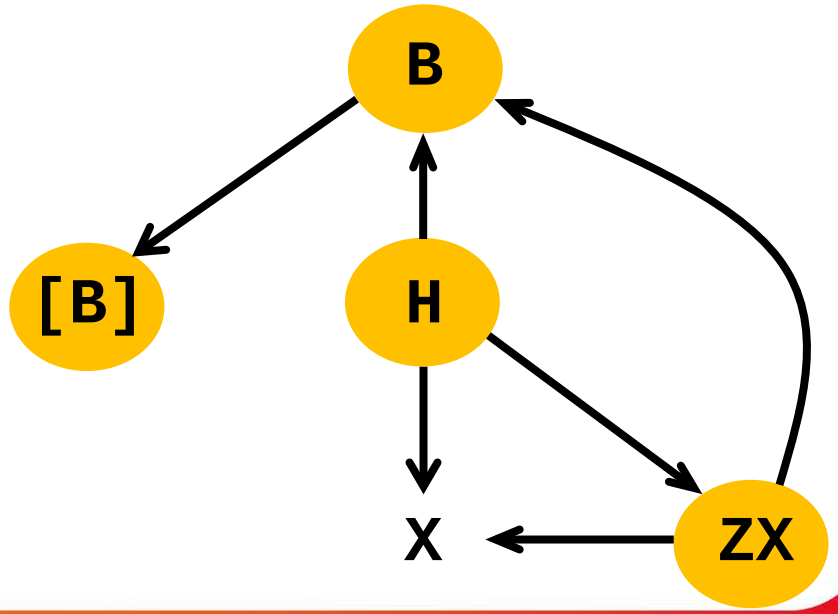


An example of Signal program and its compilation

```
(  X := IN default ZX-1      stream func
| ZX := X$1 init 0           stream func
|  B := (ZX ≤ 0)             stream func
| IN ^= (when B)             clock eqn
|  H ^= B ^= X ^= ZX )        clock eqn
```

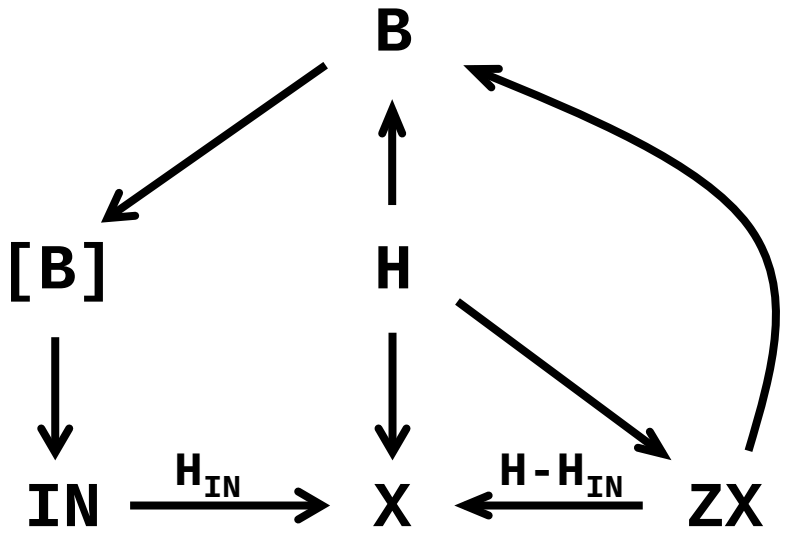
```
[B]: when B

case B false
```



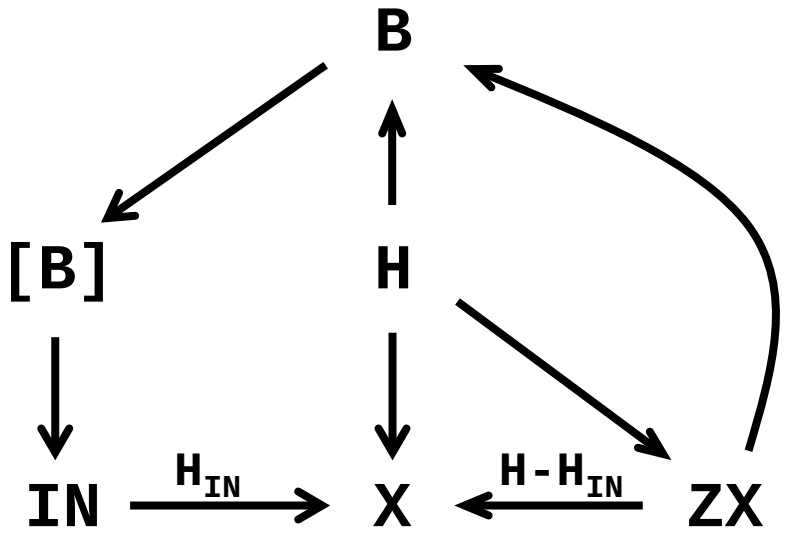
An example of Signal program and its compilation

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= (when B)
|  H ^= B ^= X ^= ZX )
```



An example of Signal program and its compilation

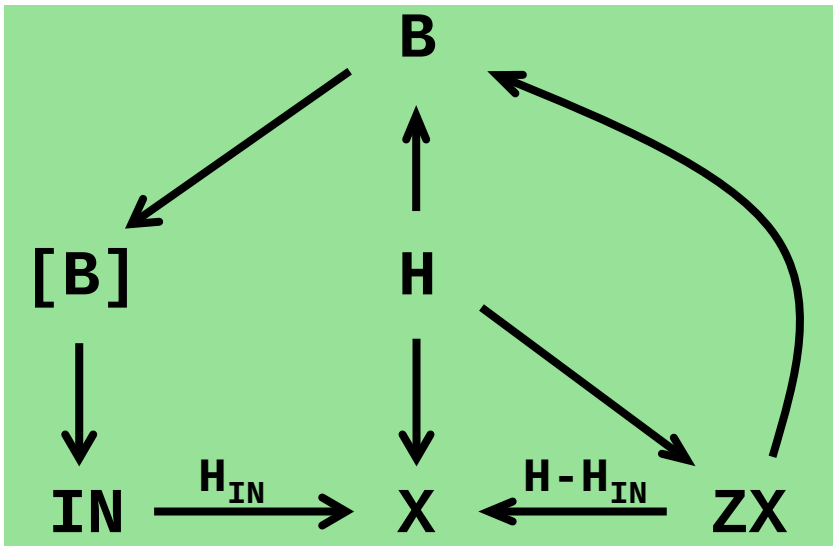
```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= (when B)
|  H ^= B ^= X ^= ZX )
```



An example of Signal program and its compilation

```
(
   (  H ^= B ^= X ^= ZX
   | IN ^= (when B) )
|
   (  X  ←  H
   | ZX  ←  H
   |  B  ← (H,ZX)
   | (when B) ←  B
   | IN  ← (when B)
   | (X  ← IN) when B
   | (X  ← ZX) when not B )
|
   (  B := (ZX ≤ 0)
   | ZX := X$1 init 0
   | (X := IN) when B
   | (X := ZX-1) when not B)
)
```

```
(  X := IN default ZX-1
| ZX := X$1 init 0
|  B := (ZX ≤ 0)
| IN ^= (when B)
|  H ^= B ^= X ^= ZX )
```



An example of Signal program and its compilation

```
(
   (  H ^= B ^= X ^= ZX
   | IN ^= (when B) )
|
   (  X  ←  H
   | ZX  ←  H
   |  B  ← (H,ZX)
   | (when B) ←  B
   | IN  ← (when B)
   | (X  ← IN) when B
   | (X  ← ZX) when not B )
|
   (  B := (ZX ≤ 0)
   | ZX := X$1 init 0
   | (X := IN) when B
   | (X := ZX-1) when not B)
)
```

Clock equations

Causality constraints

Computation actions

An example of Signal program and its compilation

```
(
   (  H ^= B ^= X ^= ZX
   | IN ^= (when B) )
|
   (  X  ← H
   | ZX  ← H
   |  B  ← (H,ZX)
   | (when B) ← B
   | IN  ← (when B)
   | (X  ← IN) when B
   | (X  ← ZX) when not B )
|
   (  B := (ZX ≤ 0)
   | ZX := X$1 init 0
   | (X := IN) when B
   | (X := ZX-1) when not B)
)
```

```
(  X := IN default ZX-1
| ZX := X$1 init 0
| IN ^= when (ZX ≤ 0) )
```

Signal compilation
is by
program rewriting

demo Polychrony later

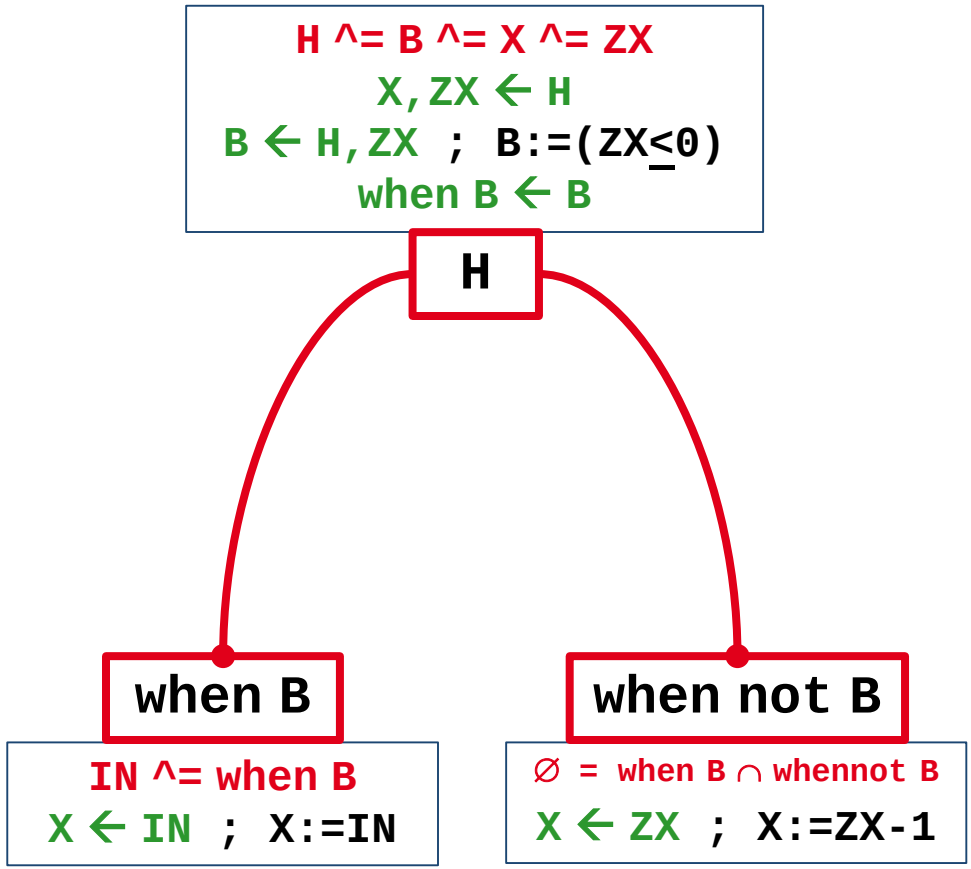
The clock and causality calculus

Intuition

Albert Benveniste and Thierry Gautier

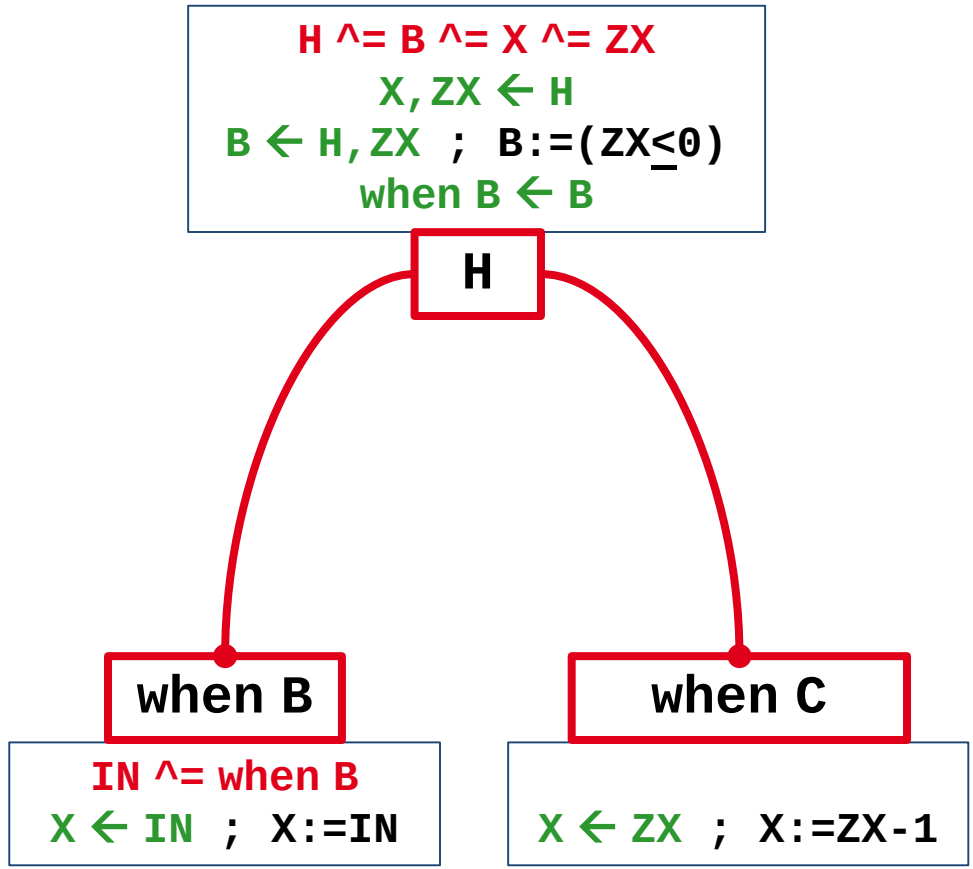
Clock and causality calculus

```
(
   (  H ^= B ^= X ^= ZX
   | IN ^= (when B) )
|
   (  X ← H
   | ZX ← H
   |  B ← H,ZX
   | (when B)← B
   | IN ← (when B)
   | (X ← IN) when B
   | (X ← ZX) when not B )
|
   (  B := (ZX ≤ 0)
   | ZX := X$1 init 0
   | (X := IN) when B
   | (X := ZX-1) when not B)
)
```



Clock and causality calculus

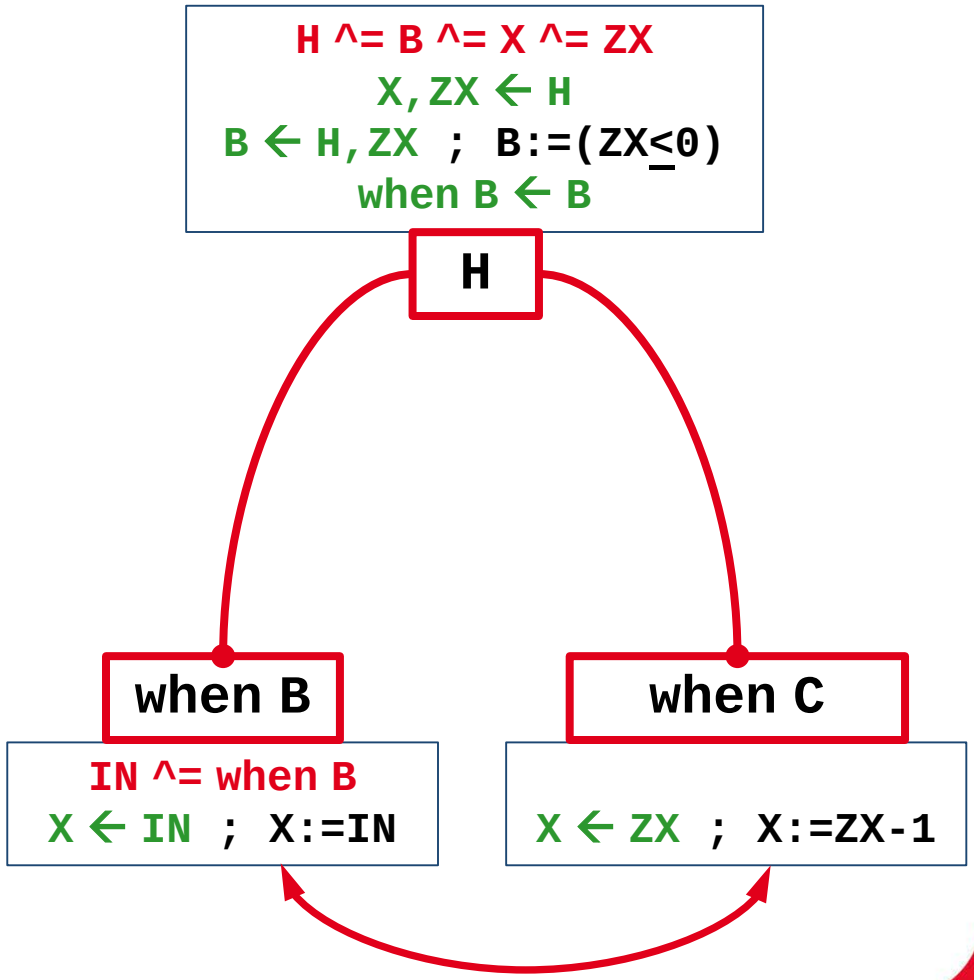
```
(
   (  H ^= B ^= X ^= ZX
   | IN ^= (when B) )
|
   (  X ← H
   | ZX ← H
   |  B,C ← H,ZX
   | (when B)← B
   | IN ← (when B)
   | (X ← IN) when B
   | (X ← ZX) when C )
|
   (  B := (ZX ≤ 0); C:=…
   | ZX := X$1 init 0
   | (X := IN) when B
   | (X := ZX-1) when C )
)
```



Clock and causality calculus

To ensure the absence of race condition, a proof obligation is added to the clock calculus:

$\varnothing$ ^= when B $\cap$ when C

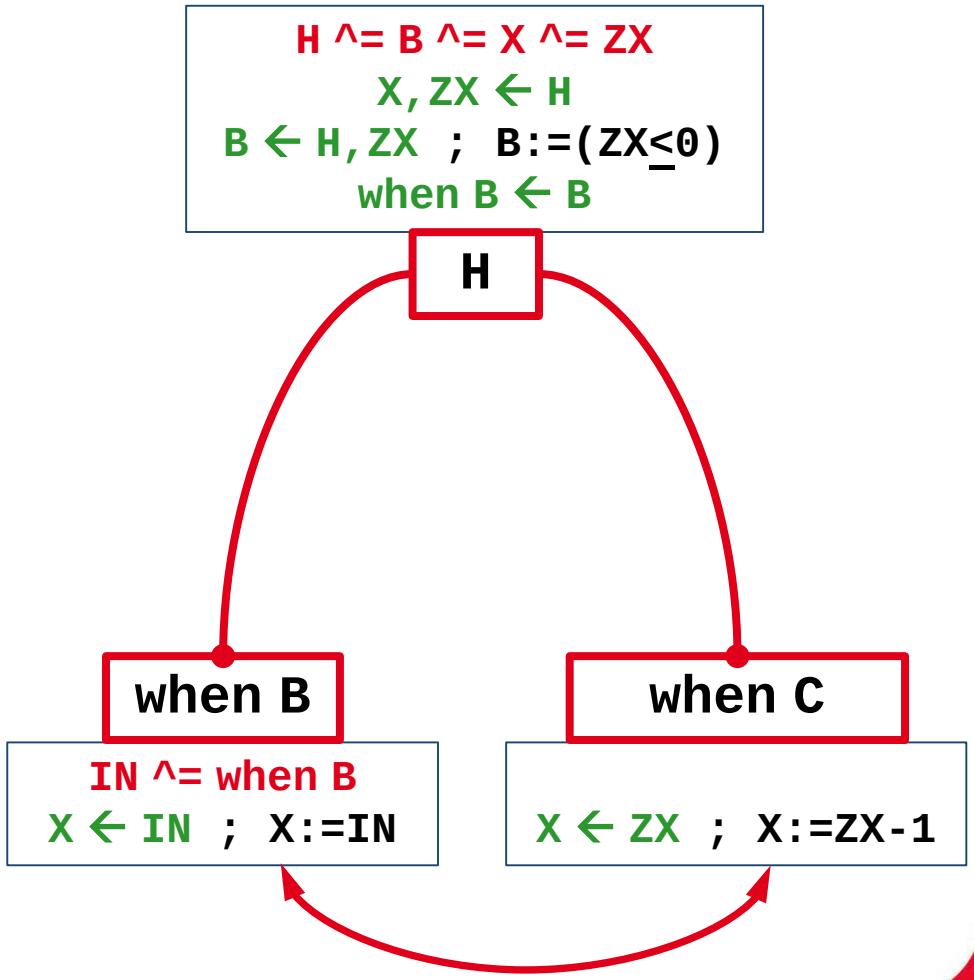


Clock and causality calculus

In general, clock equations originate from:

- the code itself
- race conditions: have them with ∅ clock
- causality circuits: have them with ∅ clock

We need to prove that the clock system is satisfiable and we must represent all solutions of it



Zooming on the causality calculus

Principles and rules of the calculus

Albert Benveniste and Thierry Gautier

Mars 2018

Clock and causality calculus

```
(
   (  H ^= B ^= X ^= ZX
   | IN ^= (when B) )
|
   (  X ← H
   | ZX ← H
   |  B ← H,ZX
   | (when B)← B
   | IN ← (when B)
   | (X ← IN) when B
   | (X ← ZX) when not B )
|
   (  B := (ZX ≤ 0)
   | ZX := X$1 init 0
   | (X := IN) when B
   | (X := ZX-1) when not B)
)
```

Rules of the causality calculus

Parallel

```
[(Y ← X) when H ]
  and
[(Y ← X) when K ]
```

implies

```
[(Y ← X) when H ^∨ K ]
```

```
(X := exp1) when H
(X := exp2) when H
```

induces the proof obligation

```
H ^= ∅
```

Sequence

```
[(Y ← X) when H ]
  and
[(Z ← Y) when K ]
```

implies

```
[(Z ← X) when H ^∧ K ]
```

```
(X ← X) when H
```

induces the proof obligation

```
H ^= ∅
```

Zooming on the clock calculus

Principles and rules of the calculus

Albert Benveniste and Thierry Gautier

The clock equations

Clock equations originate from:

- the code itself
- race conditions: have them with ∅ clock
- causality circuits: have them with ∅ clock

Wanted: a **clock hierarchy,** leading to code with nested ifs

Clocks and clock equations

1. ∅ **(nil)**; no **“top”**
2. **H ^= K**
3. **H $^{\wedge}$ K, H $^{\vee}$ K**
4. **H ^- K** (**not K** by abuse)
5. **when pred(X,Y,…)**

The clock equations

For the classes 1—4 of eqns a near-Boolean calculus applies:

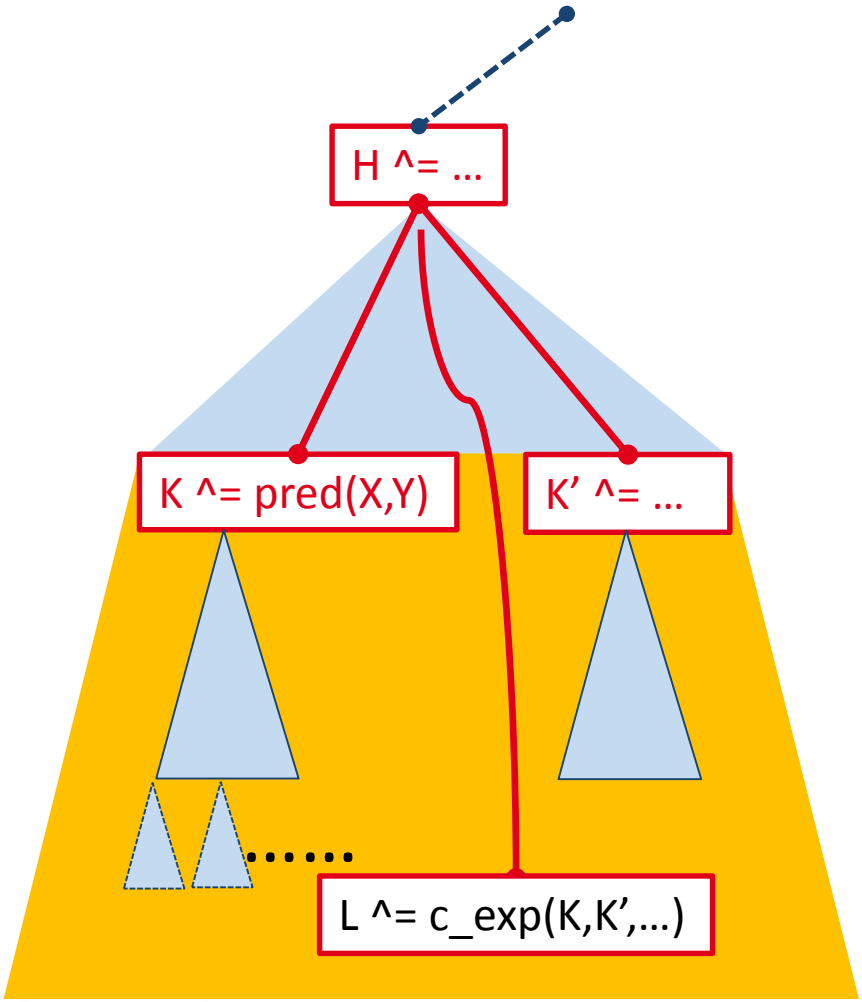
- the only difference is that no top exists

Class 5 is special:
`when pred(X,Y,…)`
is a predicate that cannot be rewritten in a different form
(**`X,Y,…`** uncontrolled)

Clocks and clock equations

1. **`∅ (nil)`**; no **“top”**
2. **`H ^= K`**
3. **`H ^∧ K, H ^∨ K`**
4. **`H ^- K`** (**not `K`** by abuse)
5. **`when pred(X,Y,…)`**

Clock hierarchization

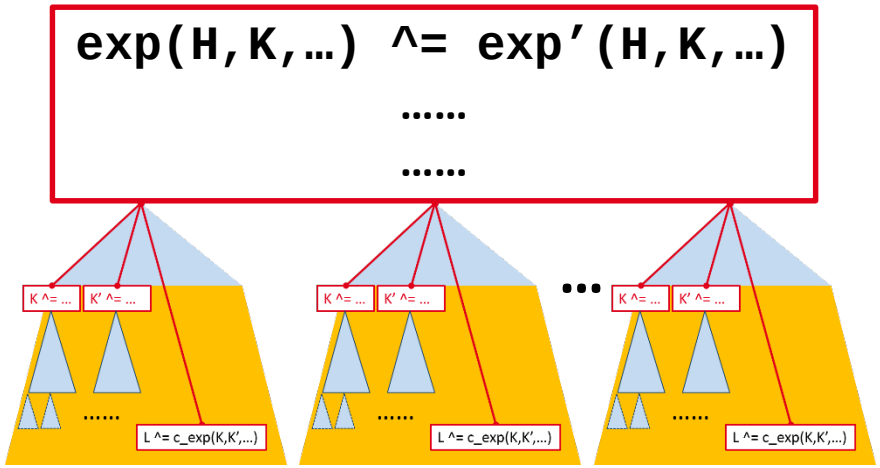


Clocks and clock equations

1. ∅ **(nil)**; no **“top”**
2. **H ^= K**
3. **H ^∧ K, H ^∨ K**
4. **H ^- K**
5. **when pred(X,Y,…)**

Clock hierarchization and calculus

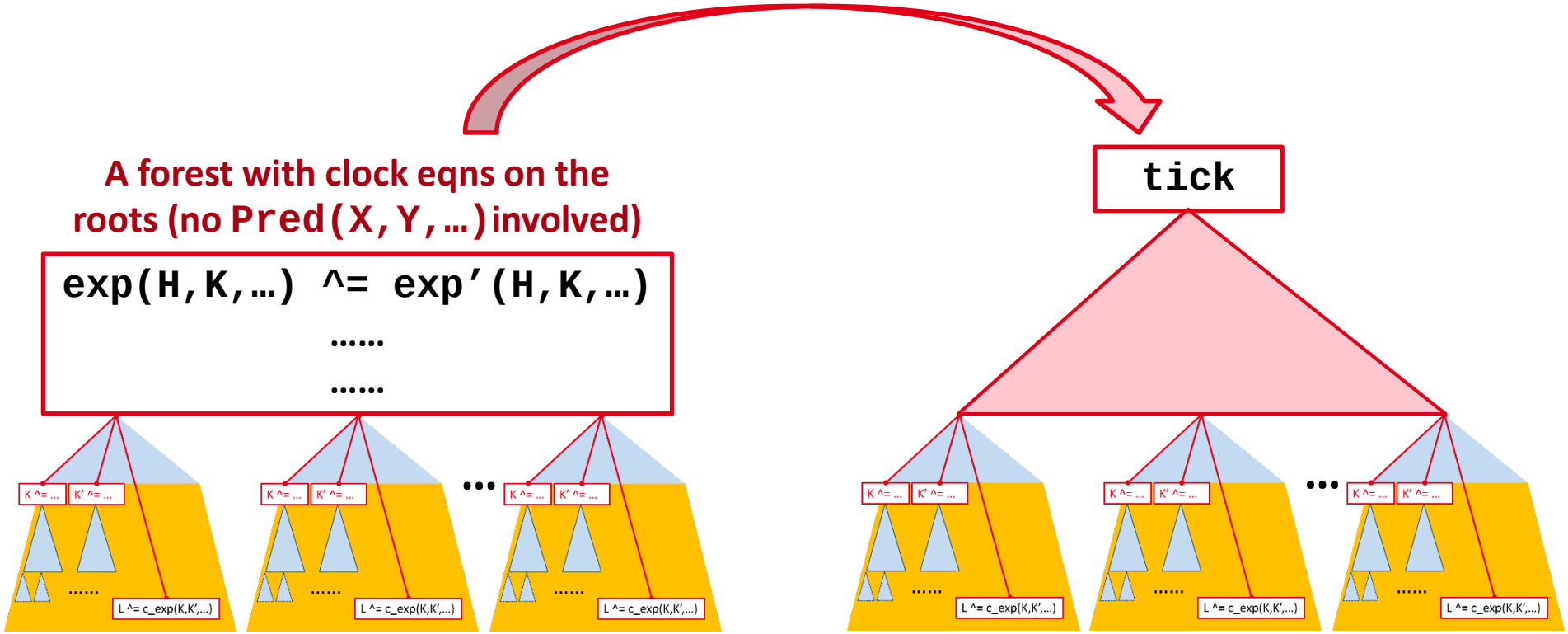
A forest with clock eqns on the roots (no `Pred(X,Y,…)` involved)



Clocks and clock equations

1. ∅ (`nil`); no "`top`"
2. `H ^= K`
3. `H ^∧ K`, `H ^∨ K`
4. `H ^- K`
5. `when pred(X,Y,…)`

Clock hierarchization and calculus



Solve the clock equations:

- check for (in)consistency
- synthesize free Boolean inputs (daemons) with clock **`tick`**
- every clock of the system is an expression involving the daemons

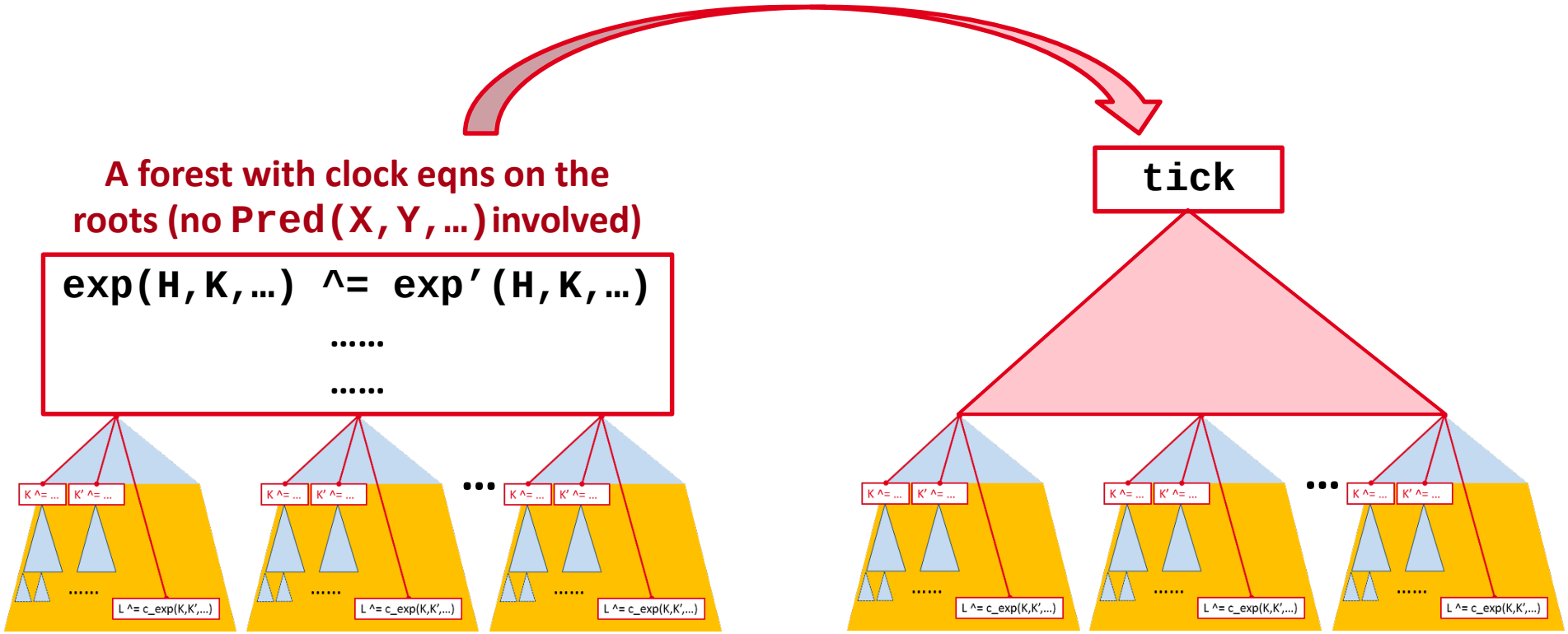
Beyond the clock calculus

Tuning the power of the engine that checks proof obligations

Albert Benveniste and Thierry Gautier

Mars 2018

Proof obligations



Solve the clock equations:

- check for (in)consistency
- synthesize free Boolean inputs (daemons) with clock **`tick`**
- every clock of the system is an expression involving the daemons

Proof obligations

Two classes of clock equations:

- Equations involving predicates **`pred(X,Y,…)`**

  `[when X>0]  ^<  [when Z<0 default when Y>5]`

- Other equations

The first class requires reasoning on **`X,Y,Z,…`**

Solve the clock equations:

- check for (in)consistency
- synthesize free Boolean inputs (daemons) with clock **`tick`**
- every clock of the system is an expression involving the daemons

Dealing with proof obligations summary

- Clock&causality calculus: background from Signal
- A *proof-obligation task* can be identified and well isolated
- This should allow reusing off-the-shelf engines:
  - SMT
  - Static analyzers
  - …

A demo of Polychrony

http://polychrony.inria.fr

- **Illustrating clock and causality calculus**
- **On-demand stepwise compilation**

A demo of some aspects of Polychrony

Repeatedly solve for X

$$aX^2 + bX + c = 0$$

where a,b,c are the input streams, by using the iterative Newton method:

$$\Delta = b^2 - 4ac, \qquad R_0 = \frac{\Delta}{2}$$

$$\forall n \geq 0:\ R_{n+1} = \frac{R_n * R_n + \Delta}{2R_n}$$

Note the unbounded internal upsampling

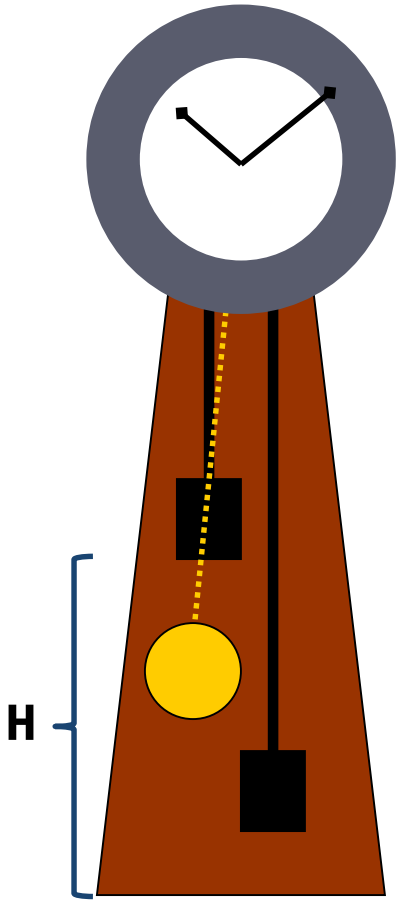
Beyond the causality calculus

Upgrading Signal to Signal+ supporting data constraints

Albert Benveniste and Thierry Gautier

Mars 2018

The venerable Signal+ clock




```
(  next T - T = -k * (next H - H)
| (next H = H - v) when not (H ≤ 0)
| (next H = IN) when (H ≤ 0)
)
```

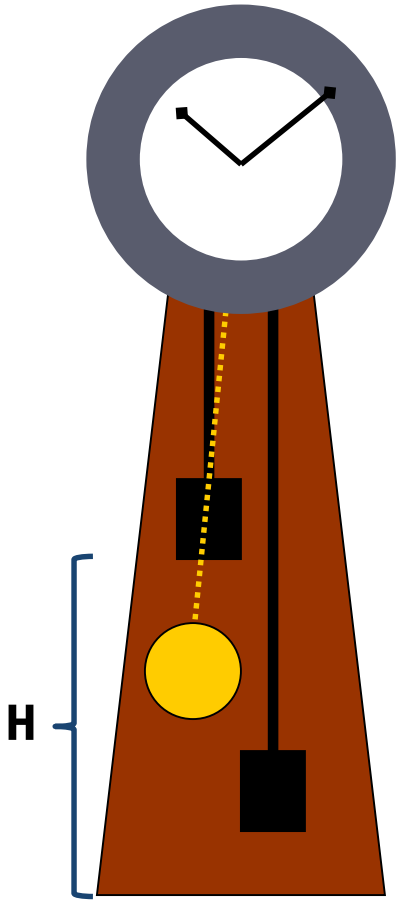
T: time

H: height of the main weight

IN: reset value for **H**

Statements: guarded equations

The venerable Signal+ clock

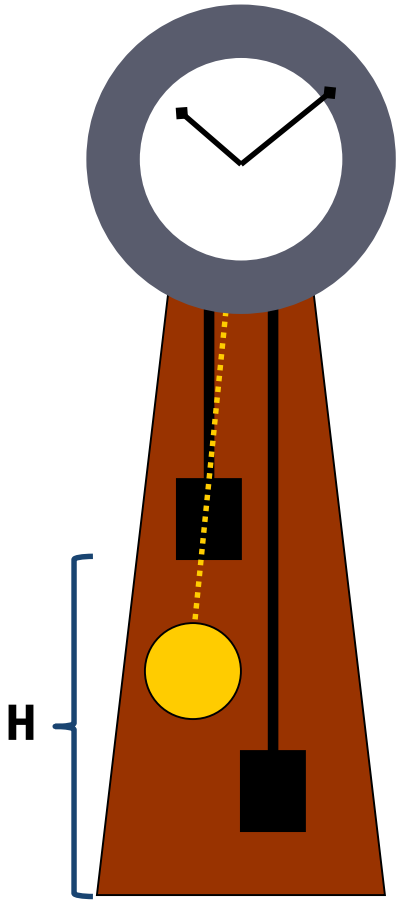



```
(   next T – T = -k * (next H - H)
| (next H = H – v) when not (H ≤ 0)
| (next H = IN) when (H ≤ 0)
)
```

Guarded equations

```
(   E1
|   E2 when not (H ≤ 0)
|   E3 when (H ≤ 0)
)
```

The venerable Signal+ clock




```
(   next T – T = -k * (next H - H)
| (next H = H – v) when not (H ≤ 0)
| (next H = IN) when (H ≤ 0)
)
```

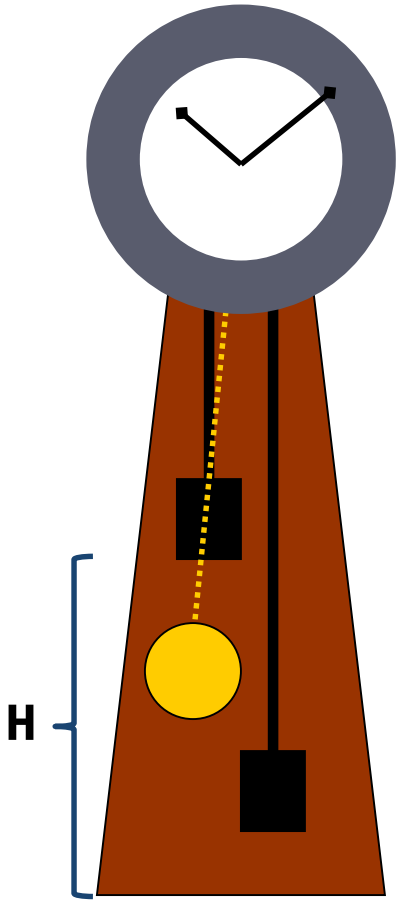
Guarded equations

```
(   E1
|   E2 when not (H ≤ 0)
|   E3 when (H ≤ 0)
)
```

Incidence graph (bi-partite, non directed)

```
(   E1 ↔ next T, next H
| (E2 ↔ next H) when not (H ≤ 0)
| (E3 ↔ next H, IN) when (H ≤ 0)
)
```

The venerable Signal+ clock




```
(   next T – T = -k * (next H - H)
| (next H = H – v) when not (H ≤ 0)
| (next H = IN) when (H ≤ 0)
)
```

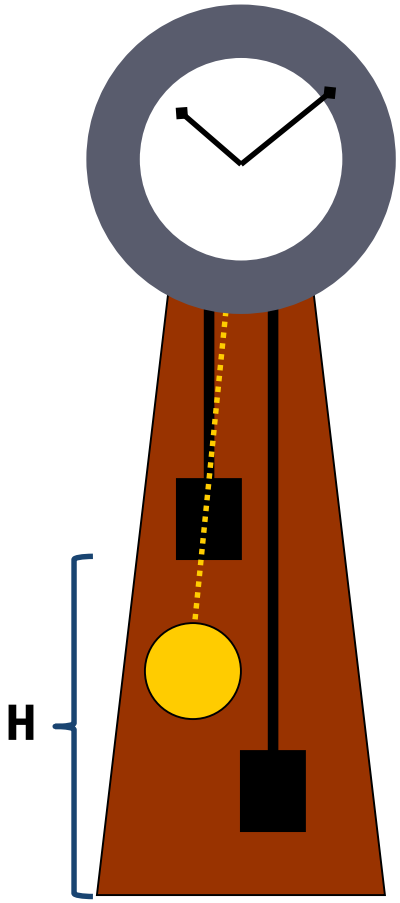
Guarded equations

```
(   E1
|   E2 when not (H ≤ 0)
|   E3 when (H ≤ 0)
)
```

Finding a guarded matching

```
(   E1 ↔ next T, next H
| (E2 ↔ next H) when not (H ≤ 0)
| (E3 ↔ next H, IN) when (H ≤ 0)
)
```

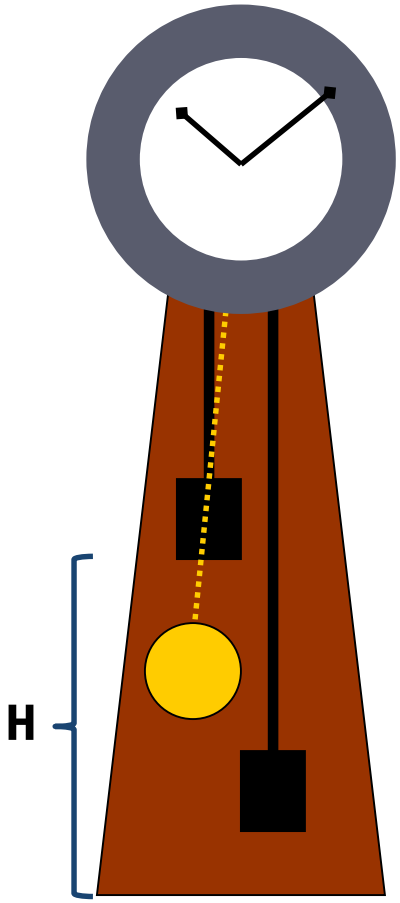
The venerable Signal+ clock



Finding a guarded matching

```
(   E1 ↔ next T, next H
| (E2 ↔ next H) when not (H ≤ 0)
| (E3 ↔ next H, IN) when (H ≤ 0)
)
```

The venerable Signal+ clock



Finding a guarded matching

```
(   E1 ↔ next T, next H
| (E2 ↔ next H) when not (H ≤ 0)
| (E3 ↔ next H, IN) when (H ≤ 0)
)
```

Yields again a scheduling

```
(   next H → E1 → next T
| (E2 → next H) when not (H ≤ 0)
| (IN → E3 → next H) when (H ≤ 0)
)
```

The rules we applied

We assumed a solver handling algebraic equations:

- **Solving system of eqns $C(x, y, z, \dots) = 0$ for $x, y, z \dots$ "scalar" variables (no tuples, no vectors)**
- **Equations possess a notion of "dimension":**
  - **if equation C=0 is itself scalar and x occurs in C, then the solver can, generically, use eqn $C = 0$ for determining x, given values for other variables**
  - **pair variables with equations defining them: $C \leftrightarrow x$**

Typical example: $x, y, z \in R$ and $C(x, y, z, \dots) = 0$ smooth

This looks like an easy generalization

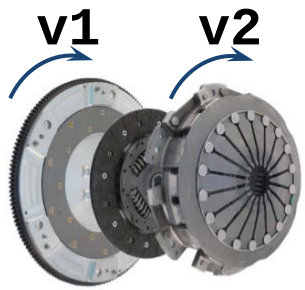
HHHmmmm??????? Too easy??????

Synchronous *specification* languages are much more difficult (but also more powerful) than synchronous languages

Example of a clutch

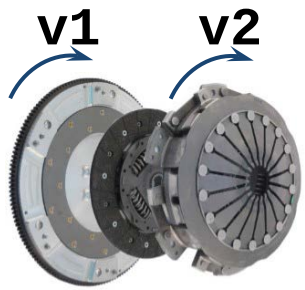
The clutch

```
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0)
 | (torque2 = 0)                    )
```



Clutch released

The clutch

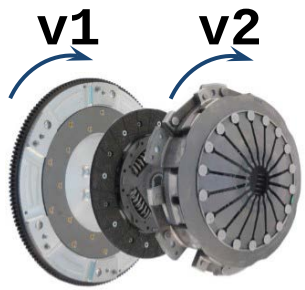


Clutch engaged

```
( next v1 = f(v1,torque1)
| next v2 = f(v2,torque2) )


|
 (
 | (v1 = v2)
 | (torque1 + torque2 = 0)                    )
```

The clutch

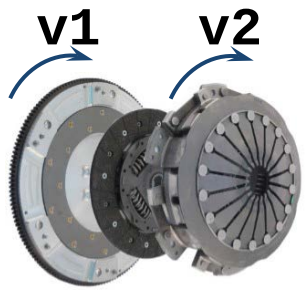


Clutch

```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0) when not Engaged
 | (torque2 = 0) when not Engaged )
|
 (
 | (v1 = v2) when Engaged
 | (torque1 + torque2 = 0) when Engaged )
)
```

At each reaction, the following must be evaluated from current states & inputs: `torque1`, `torque2`, `next v1`, `next v2`

The clutch



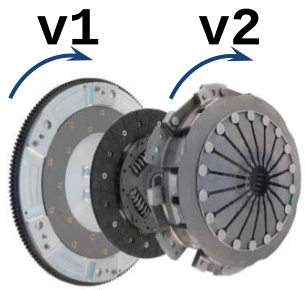
Clutch

```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0) when not Engaged
 | (torque2 = 0) when not Engaged )
|
 (
 | (v1 = v2) when Engaged
 | (torque1 + torque2 = 0) when Engaged )
)
```

Two problems:

- **v1 = v2** constrains the memories
- Engaged mode : 4 variables but only 3 equations

The clutch



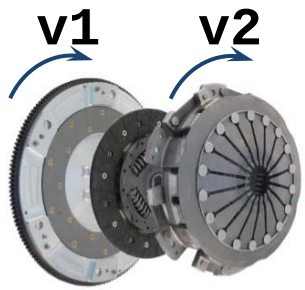
Clutch

```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0) when not Engaged
 | (torque2 = 0) when not Engaged )
|
 ( (next v1 = next v2) when Engaged
 | (v1 = v2) when Engaged
 | (torque1 + torque2 = 0) when Engaged )
)
```

Case clutch engaged at previous reaction:

adding the blue eqn is legitimate and gives the missing equation (*index reduction*)

The clutch



Clutch

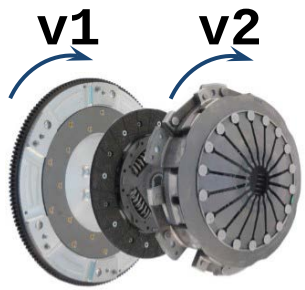
```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0) when not Engaged
 | (torque2 = 0) when not Engaged )
|
 ( (next v1 = next v2) when Engaged
 | (v1 = v2) when Engaged
 | (torque1 + torque2 = 0) when Engaged )
)
```

Case clutch *not* engaged at previous reaction:

adding the blue eqn is legitimate and gives the missing equation

the green eqn is falsified

The clutch



Clutch

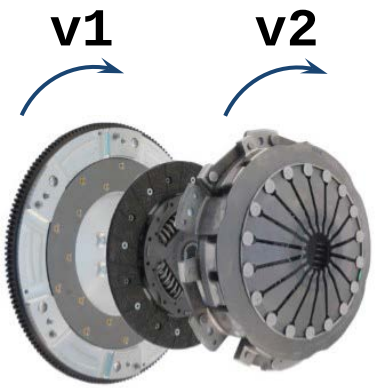
```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0) when not Engaged
 | (torque2 = 0) when not Engaged )
|
 ( (next v1 = next v2) when Engaged
 |
 | (torque1 + torque2 = 0) when Engaged )
)
```

Case clutch *not* engaged at previous reaction:

adding the blue eqn is legitimate and gives the missing equation

the green eqn is falsified: we remove it

The final code for the clutch




```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (next v1 = next v2)
 | (torque1 + torque2 = 0) )
)
```

engaging

released

```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (torque1 = 0)
 | (torque2 = 0) )
)
```

engaged

```
(
 ( next v1 = f(v1,torque1)
 | next v2 = f(v2,torque2) )
|
 ( (next v1 = next v2)
 | (v1 = v2)
 | (torque1 + torque2 = 0) )
)
```

Conclusion

- Clock and causality calculus is a background from Signal
- The notion of index is a background from DAE and Modelica
- It seems feasible to upgrade Signal to handle “numerical constraints”
- Our guess is that Signal compilation techniques should be borrowable
- In particular, it should be possible to avoid enumerating modes
- Is this idea any useful??

Thanks