A Protocol for Loosely Time-Triggered Architectures — LTTA

Albert Benveniste(1), Paul Caspi(2),
Paul Le Guernic(1), Hervé Marchand(1),
Jean-Pierre Talpin(1), Stavros Tripakis(2)

(1) Irisa/Inria, Rennes , (2) Verimag, Grenoble

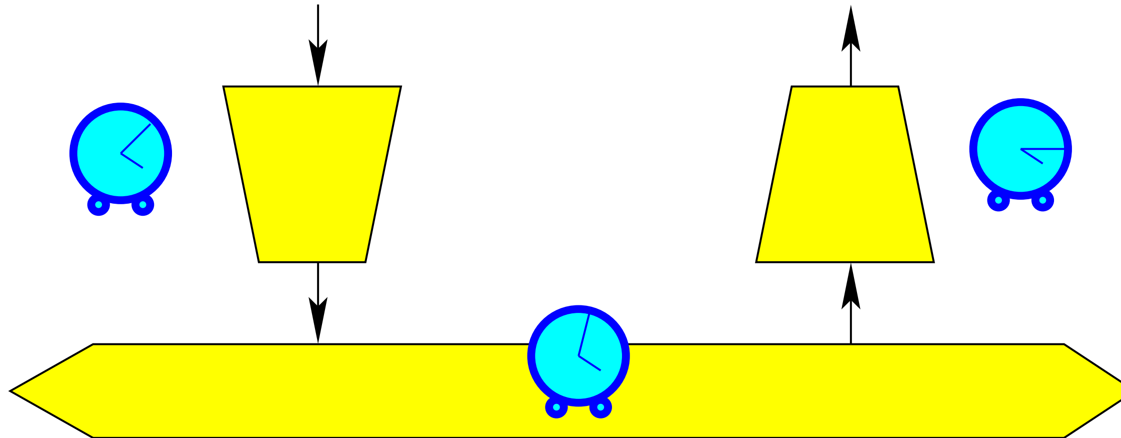
Emsoft'2002

Contents

LTTA:

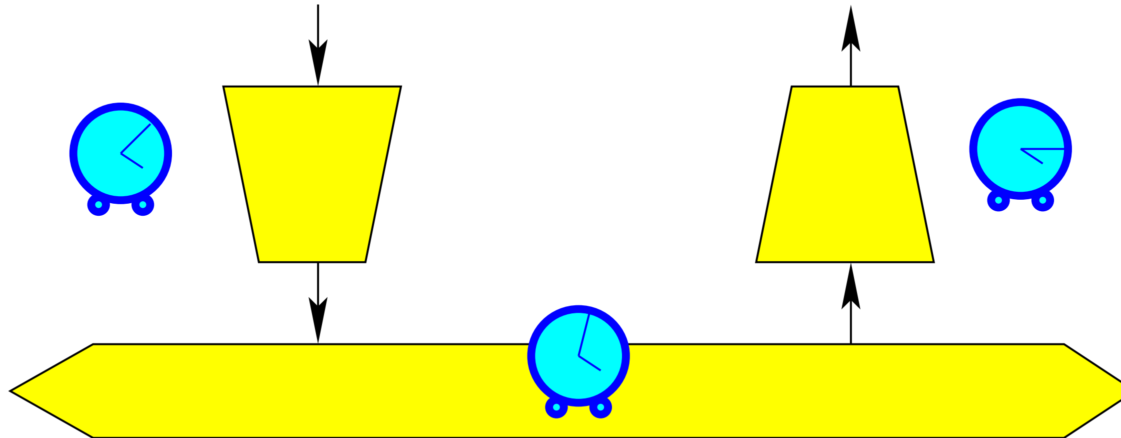
1. **what, where, why**
2. **problem**
3. **solution**
4. **analysis**

LTTA bus (cf. Airbus)



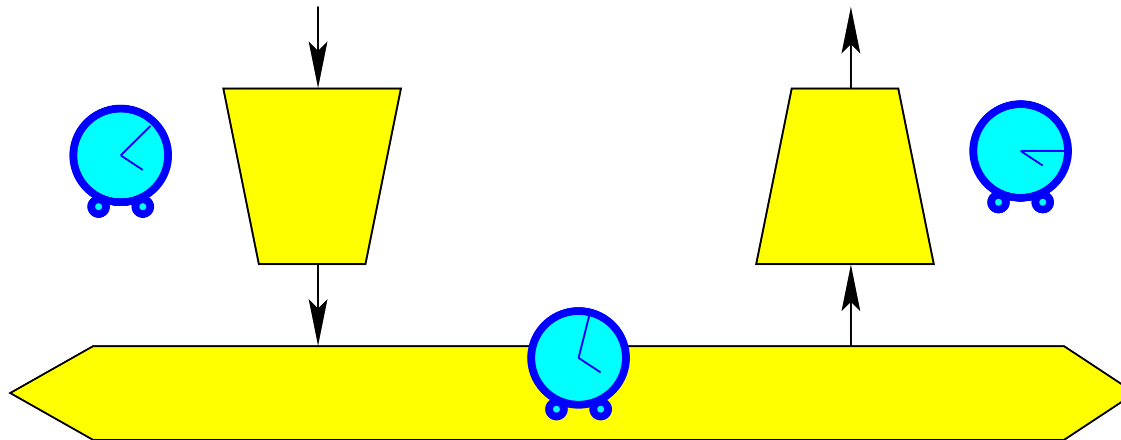
the writer's buffer is periodic
the bus is periodic
the reader's buffer is periodic

LTTA bus (cf. Airbus)



the writer's buffer is periodic
the bus is periodic
the reader's buffer is periodic
values are sustained in writer/bus/reader
clocks are not physically synchronized

LTTA bus (cf. Airbus)



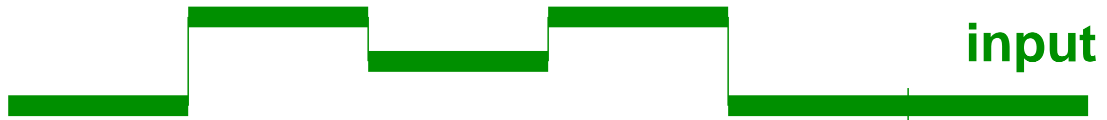
the writer's buffer is periodic
the bus is periodic
the reader's buffer is periodic
values are sustained in writer/bus/reader
clocks are not physically synchronized
a leightweight, flexible architecture

Contents

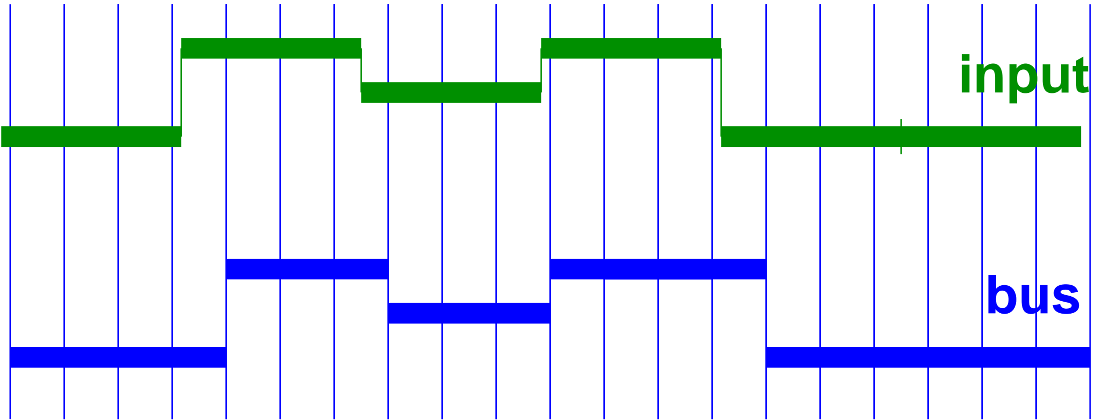
LTTA:

1. **what, where, why**
2. **problem**
3. **solution**
4. **analysis**

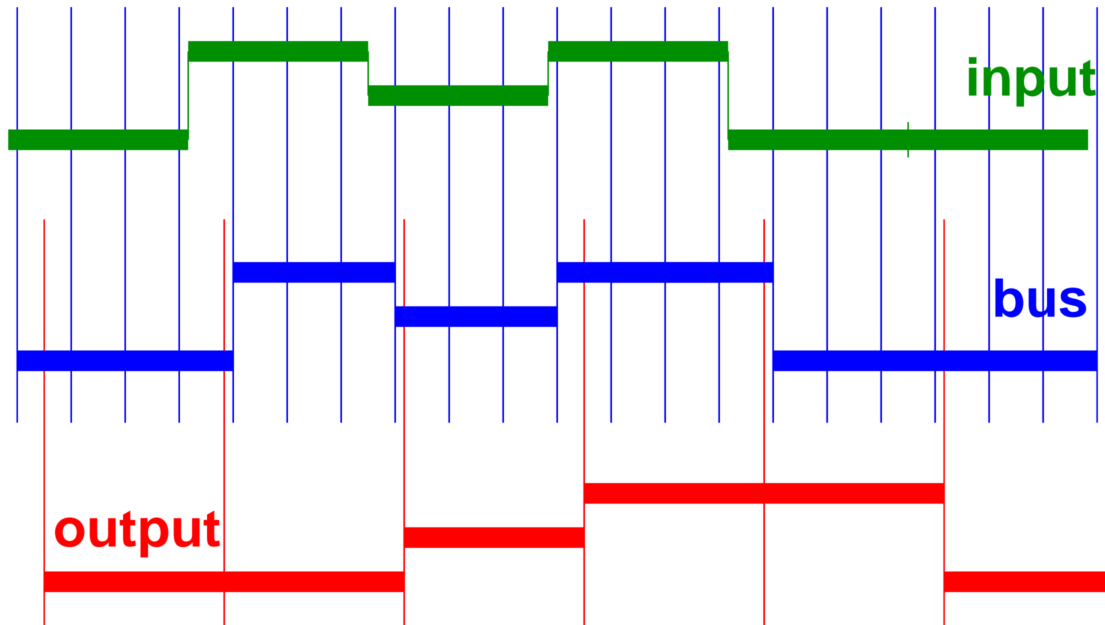
transmitting a signal over LTTA



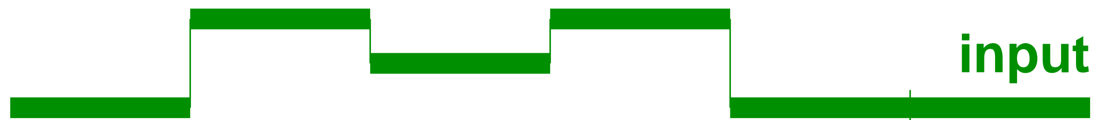
transmitting a signal over LTTA

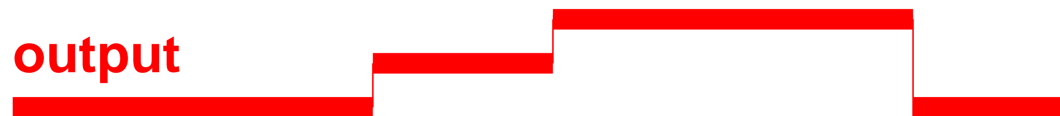


transmitting a signal over LTTA



transmitting a signal over LTTA





it can lose or duplicate data,

but boundedly so

LTTA bus can lose or duplicate data, but boundedly so

LTTA bus can lose or duplicate data, but boundedly so

This is acceptable for distributed low-level sampled-data control, since control design methods are robust enough to accomodate for this, thanks to continuity and stability of the closed-loop system.

LTTA bus can lose or duplicate data, but boundedly so

This is acceptable for distributed low-level sampled-data control, since control design methods are robust enough to accomodate for this, thanks to continuity and stability of the closed-loop system.

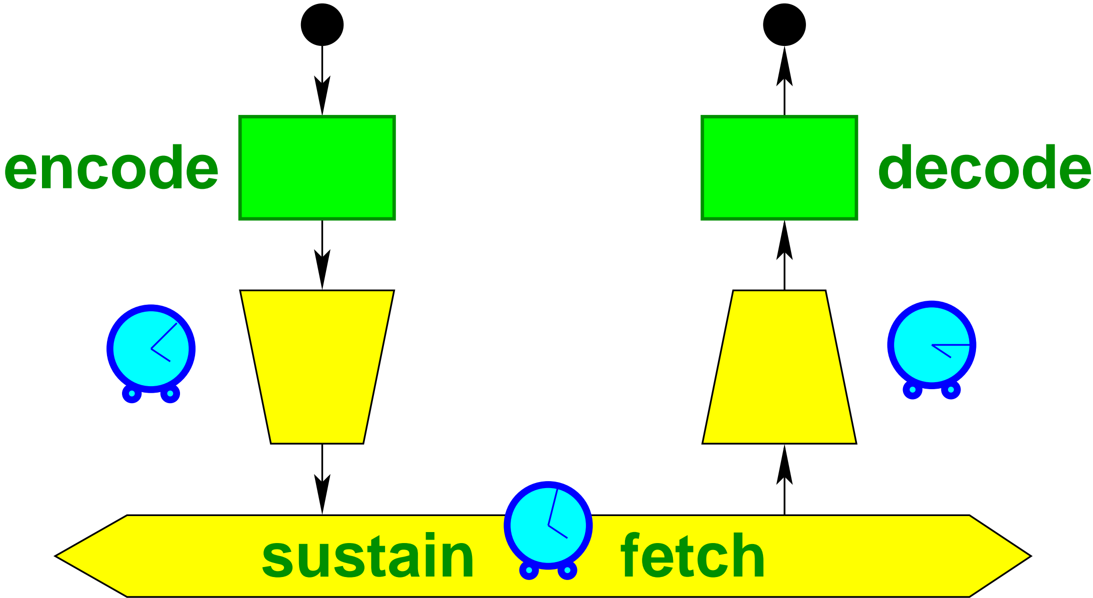
But this may be a problem to implement distributed discrete control of operating modes, or protection control.

Contents

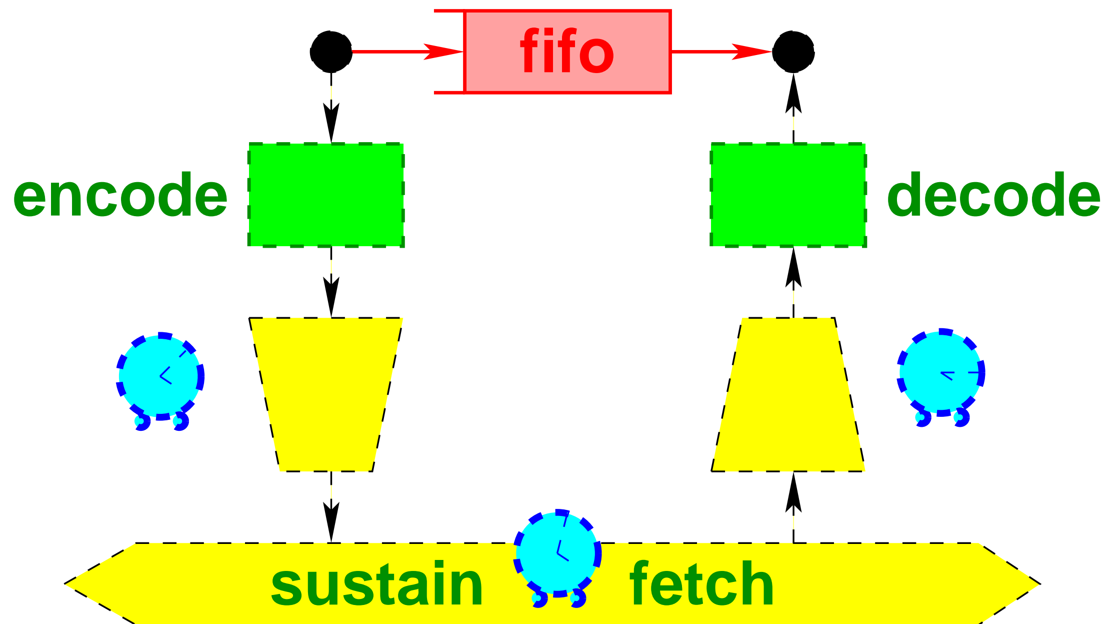
LTTA:

1. what, where, why
2. problem
3. solution
4. analysis

A protocol on the top of LTTA

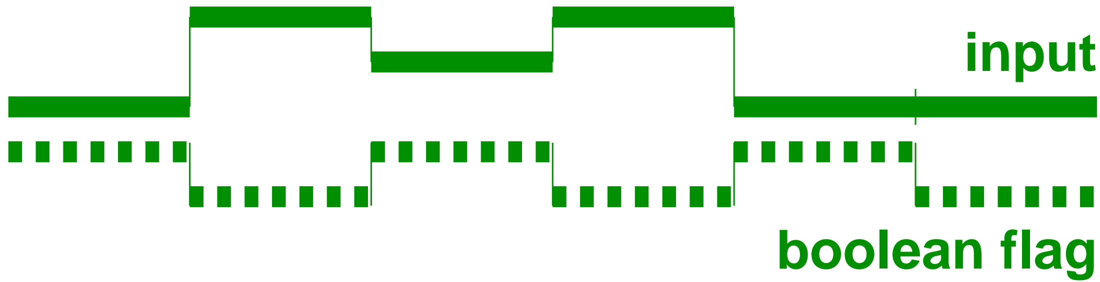


A protocol on the top of LTTA

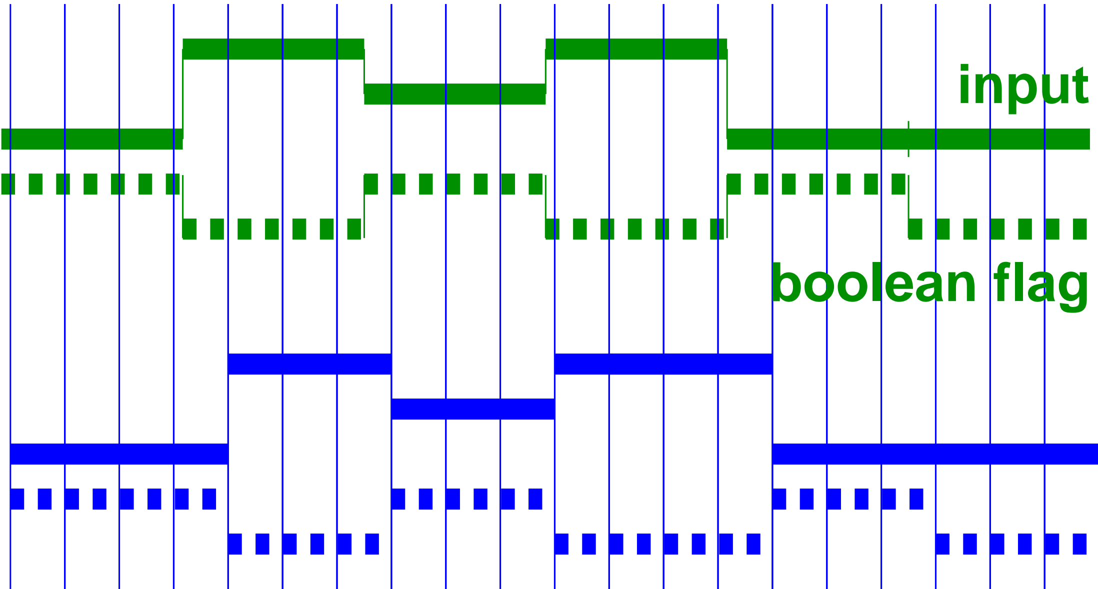


when encapsulated by this protocol, the medium behaves like a point-to-point network of FIFO channels: enough to apply the Benveniste-Caillaud technique for distributed implementation of synchronous programs

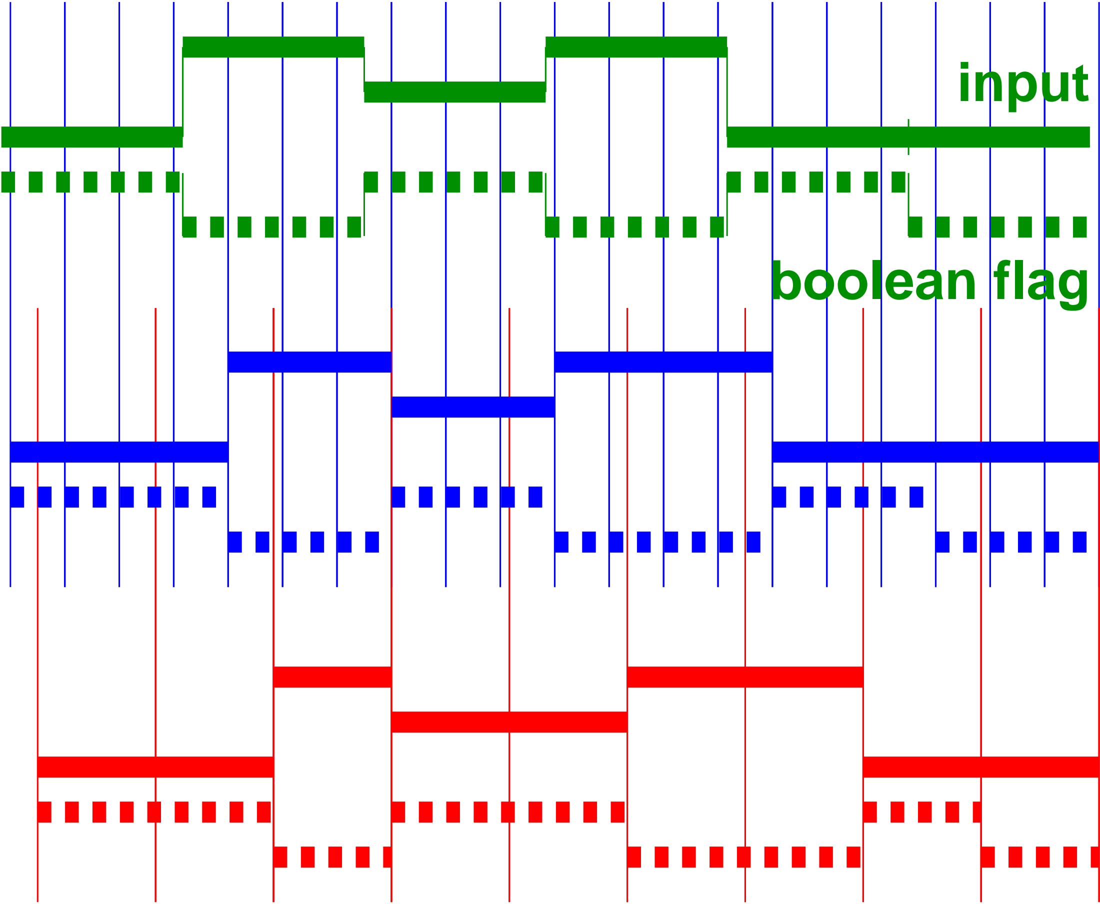
behaviour of the protocol



behaviour of the protocol

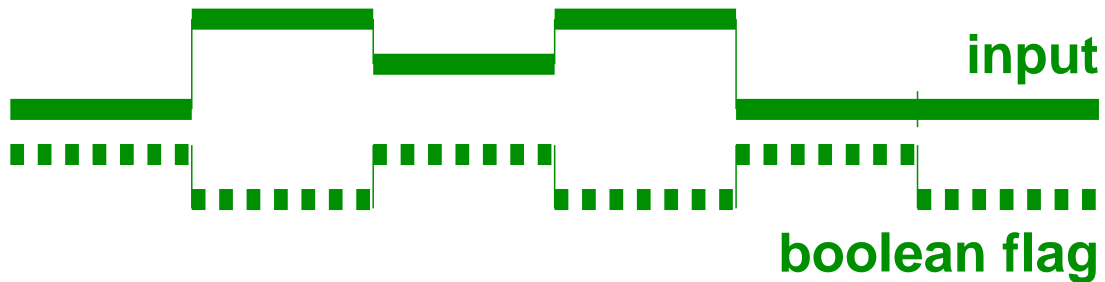


behaviour of the protocol

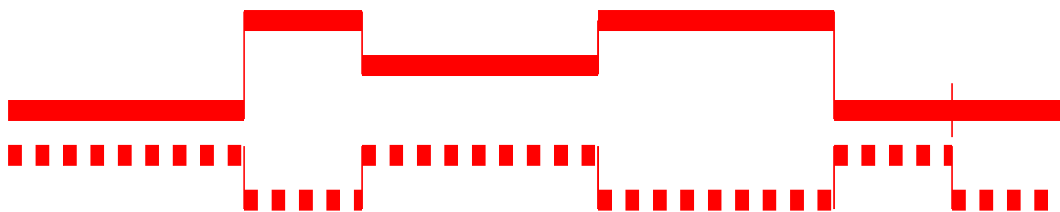


$$w \geq b \quad \textbf{and} \quad \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b}$$

behaviour of the protocol



THEOREM: the protocol behaves as a bundle of FIFO channels, with variable but bounded delay



$$w \geq b \quad \textbf{and} \quad \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b}$$

Contents

LTTA:

1. what, where, why
2. problem
3. solution
4. analysis

how to prove the theorem?

how to prove the theorem?

1. by " brainual " proof (paper)

extends to almost periodic clocks (robustness)

how to prove the theorem?

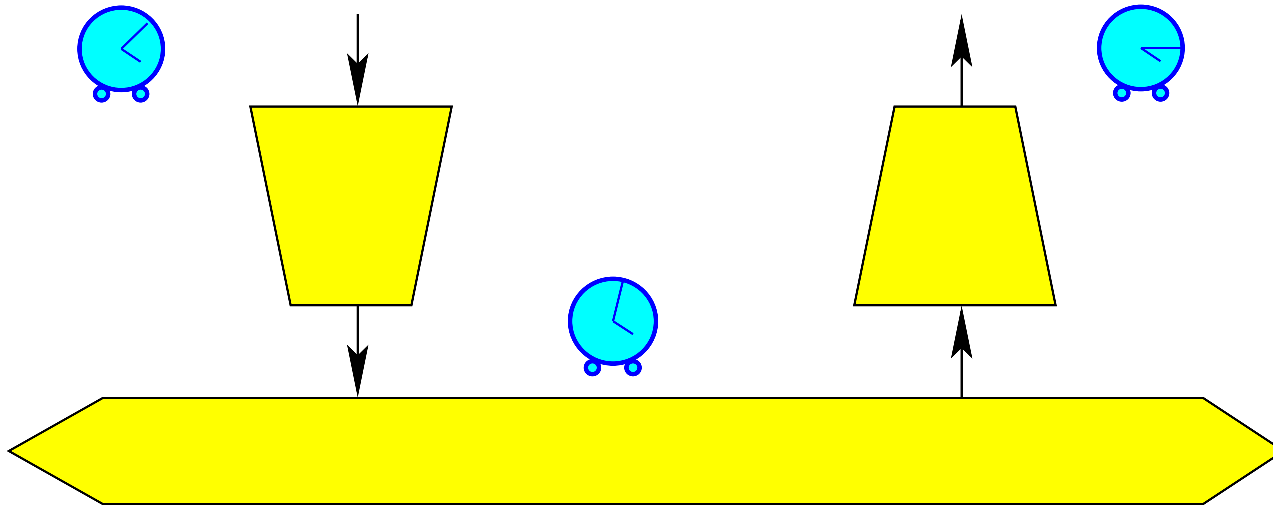
1. by “ brainual ” proof (paper)

   extends to almost periodic clocks (robustness)

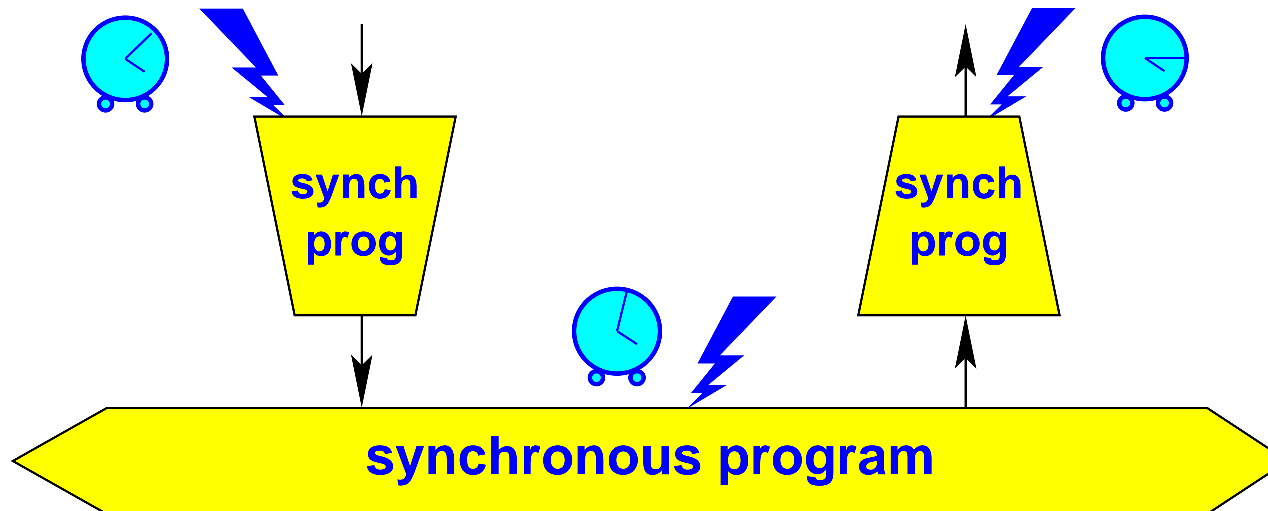
2. (almost) automatically

   by formal analysis of a distributed asynchronous system using synchronous languages!

Principle of the automatic proofs

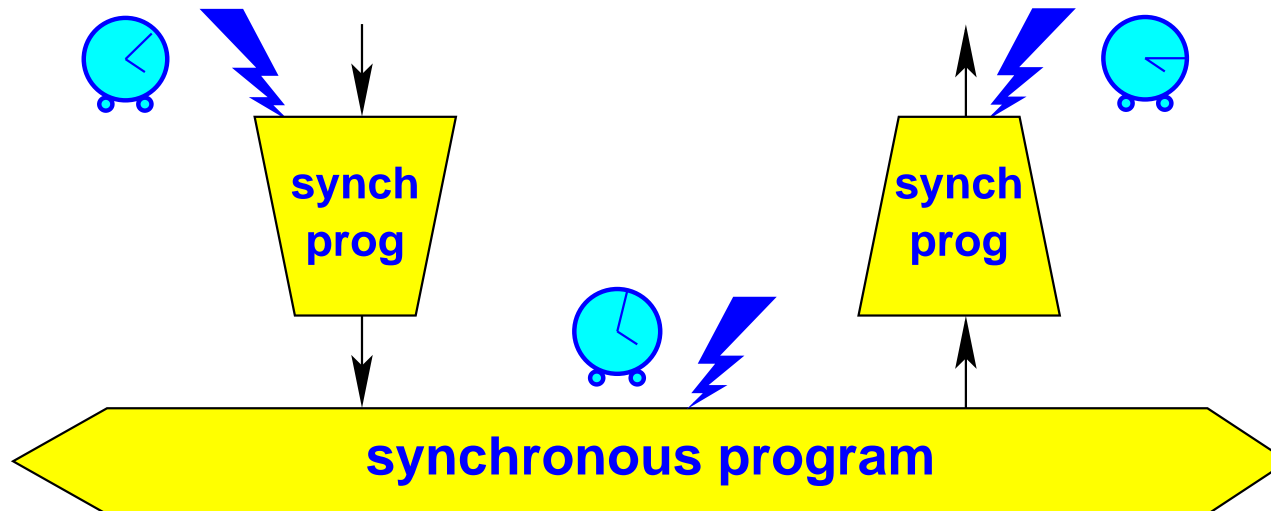


Principle of the automatic proofs



how to abstract the metric condition
$[w \geq b] \wedge [\lfloor \frac{w}{b} \rfloor \geq \frac{r}{b}]$ **into a logical one ?**

Principle of the automatic proofs

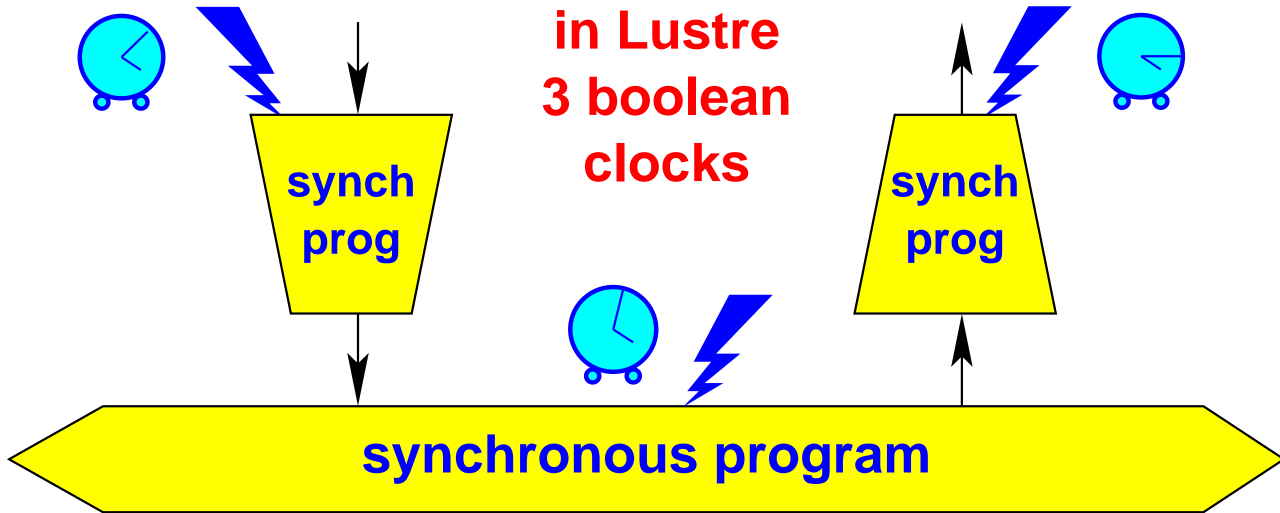


how to abstract the metric condition $[w \geq b] \wedge [\left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b}]$ into a logical one ?

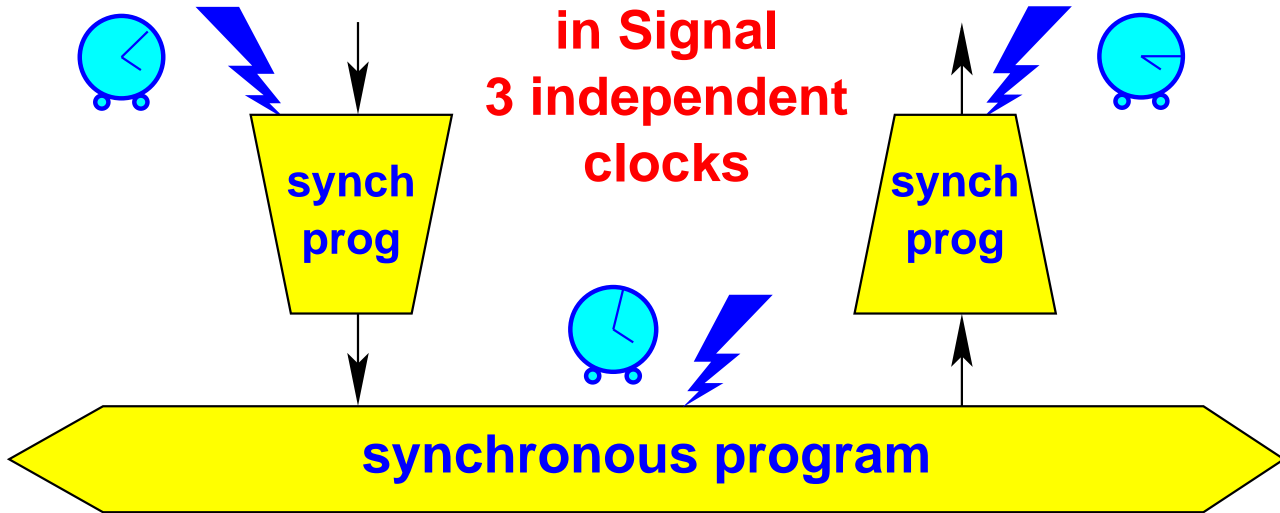
$[w \geq b]$: **never two t^{w} between two t^{b}**

$[\left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b}]$: **more difficult, but feasible**

Principle of the automatic proofs



Principle of the automatic proofs



The Lustre proof

```
const n = 3;   -- the input is a bit stream of width 3

node writer(x : bool^n) returns (xw: bool^n; bw: bool);
let
    bw = true -> pre not bw;
    xw = x;
tel

const init = false^n;

node reader(x: bool^n; b: bool) returns (cro: bool; xr: bool^n);
let
    cro = not (b = (false -> pre b));
    xr  = if cro then x
          else (init -> pre xr);
tel

node bus(xw: bool^n; bw: bool) returns (xr: bool^n; br: bool);
let
    xr, br = (xw, bw);
tel

node faster(cb, cw: bool) returns (prop: bool);
var w_before_b: bool;
let
    w_before_b = if cw then true
                 else if cb then false
                      else (false -> pre w_before_b);
    -- tells that there is an unmatched cw
    prop = not (cw and (false -> pre w_before_b));
-- this node implements (??)
tel

node firstafter(cb, cw: bool) returns (cbw: bool);
var waiting: bool;
let
    cbw = cb and (false -> pre waiting) ;
    waiting = if cw then true
              else if cbw then false
                   else (false -> pre waiting);
-- this node implements (??)
tel

node vecteq(xw: bool^n; xr: bool^n) returns (prop: bool);
var aux: bool^(n+1);
let
    aux[0] = true;
    aux[1..n] = aux[0..n-1] and (xr = xw);
    prop = aux[n];
tel

node compare(cw: bool; xw: bool^n; xr: bool^n) returns (prop: bool);
var equal: bool; last: bool^n; unmatched: bool;
let
    last = if equal then xw else (init -> pre last);
    -- stores the value to be matched
    equal = vecteq(xr, (init -> pre last));
    -- tells whether the value to be matched is actually matched
    unmatched = if cw and not (true -> pre equal) then true
                else if equal then false
                     else false -> pre unmatched;
    -- tells that there are two values waiting for match
    prop = not(cw and (false -> pre unmatched));
    -- a new value should not arrive while two values are waiting for match
tel

node verif(cw, cb, cr: bool; (x: bool^n) when cw)
returns (prop: bool; xw, xr, xro: bool^n; bw, br: bool; cro: bool );
let
    xw, bw = if cw then current writer(x)
             else ((init, false) -> pre(xw, bw));
    xr, br =  if cb then current bus((xw, bw) when cb)
             else ((init, false) -> pre(xr, br));
    cro, xro =if cr then current reader((xr, br) when cr)
             else ((false, init) -> pre(cro, xro));

    prop = compare(cw, xw, xro);

    assert faster(cb, cw) and faster(cr, firstafter(cb, cw));
    -- these assertions implement (??) and (??)
    assert #(cw, cb, cr);
    -- so as not to get bored by simultaneous clocks
tel

(*
moucherotte% lesar albert2.lus verif
--Pollux Version 2.0
TRUE PROPERTY
moucherotte%
)
```

The Signal proof

```
process protocol = (? boolean xw; event cw, cb, cr ! boolean xr , inv)
    (| (xb, bb, sbw) := bus (xw, writer(xw,cw), cb)       % writer + bus %
     | (xr, br, sbb)  := reader (xb, bb, cr)              % reader %
     | cb ^= sbw default cb                               % condition (??) %
     | cr ^= (when switched(sbb)) default cr              % condition (??) %
     | xok := fifo_2 (xw)                                 % fifo_2 satisfies (??) %
     | inv  := equal (xok, xr)                            % tests if xok=xr %
     |) where boolean bw, xb, bb, sbw, sbb, br, xok;

    process writer = (? boolean xw; event cw ! boolean bw)
        (| bw ^= xw ^= cw
         | bw := not (bw$1 init true)
         |);                                              % bw: boolean flag %
   process bus = (? boolean xw, bw; event cb ! boolean xb, bb, sbw)
        (| (xb, bb, sbw) := buffer (xw, bw, cb) |);
   process reader = (? boolean xb, bb; event cr ! boolean xr, br, sbb)
        (| (yr, br, sbb) := buffer (xb, bb, cr) | xr := yr when switched (br) |)
        where boolean yr; end;                           % switched(br) validates xr %

   process switched = (? boolean b ! boolean c)
        (| zb := b$1 init true | c := (b and not zb) or (not b and zb) |)
        where boolean zb; end;                           % c=true when b alternates %

   process buffer = (? boolean x, b ; event c ! boolean bx, bb, sb)
        (| (sx, sb) := shift_2 (x, b) | (bx, bb) := current_2 (sx, sb, c) |)
        where boolean sx; end;                           % delays, sustains, filters %

   process shift_2 = (? boolean x, b ! boolean sx, sb)   % see shift_1 %
        (| (sx, sb) := current_2 (x, b, ^sb) | interleave (x, sx) |);
      process current_2 = (? boolean wx, wb; event c ! boolean rx, rb)
           (| rx := (wx cell c init false) when c
            | rb := (wb cell c init true) when c |);     % see current_1 %
      process interleave = (? boolean x, sx ! )
           (| x ^= when b | sx ^= when not b | b := not (b$1 init false) |)
           where boolean b; end;                         % x and sx interleave %

   process equal = (? boolean y, z ! boolean inv)
        (| i := (y and z) or (not y and not z) default inv
         | inv := i $1 init true
         |); where boolean i; end;                       % tests if y=z %

   process fifo_2 = (? boolean x ! boolean xok )
        (| xok := shift_1(shift_1(x)) |);
      process shift_1 = (? boolean x ! boolean sx)       % x,sx satisfy (??) %
        (| sx := current_1 (x, ^sx) | interleave (x, sx) |);
         process current_1 = (? boolean wx; event c ! boolean rx)
           (| rx := (wx cell c init false) when c |);   % current triggered by c %

end;
```

```
Sigali:
————————————————
set_reorder(1);
read("protocol.z3z");
read("Creat_SDP.z3z");
read("Verif_Determ.z3z");
POSSIBLE(B_False(S,inv)); → resultat False
Always(B_True(S,inv)); → resultat True
————————————————
```

CONCLUSION

LTTA architectures (such as in use, e.g., at Airbus) can be made GALS-like

this allows for the distributed deployment of synchronous programs

this is probably a particular case of a more general theory of “ correct distributed deployments ”, currently under study