A Protocol for Loosely Time-Triggered Architectures — LTTA

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

\textsuperscript{(1)} Irisa/Inria, Rennes , \textsuperscript{(2)} Verimag, Grenoble

Emsoft’2002
LTTA:

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

Contents
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 lightweight, flexible architecture
Contents

LTTA:

1. what, where, why
2. problem
3. solution
4. analysis
transmitting a signal over LTTA
transmitting a signal over LT TA
transmitting a signal over LTTA

input

bus

output
transmitting a signal over LTTA

input

output

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 accommodate 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 accommodate 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

LT TA:

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

input

boolean flag
behaviour of the protocol
behaviour of the protocol

\[ w \geq b \quad \text{and} \quad \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b} \]
behaviour of the protocol

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

\[ w \geq b \quad \text{and} \quad \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b} \]
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 
\([w \geq b] \land \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b}\) into a logical one?
Principle of the automatic proofs

how to abstract the metric condition

\[ [w \geq b] \land \left[ \frac{w}{b} \geq \frac{r}{b} \right] \] into a logical one?

\[ [w \geq b] \] : never two \( t^W \) between two \( t^B \)

\[ \left[ \frac{w}{b} \geq \frac{r}{b} \right] \] : more difficult, but feasible
Principle of the automatic proofs

in Lustre
3 boolean clocks

synchronous program
Principle of the automatic proofs

in Signal
3 independent
clocks

synch prog

synch prog

synchronous program

how to abstract the metric condition \[ w^2 \]

between two

more difficult, but feasible
const n = 3;  the input is a bit stream of width 3

node writer(x : boolean[n]) returns (xw : boolean[n]; bw : boolean); let
    bw = true \rightarrow \text{pre not bw};
    xw = x;
end

const init = false^n;

node reader(x : boolean[n]; b : boolean) returns (cro : boolean; xr : boolean[n]); let
    cro = not (b = (false \rightarrow \text{pre b}));
    xr = if cro then x
        else (init \rightarrow \text{pre xr});
end

node buf(xw : boolean[n]; bw : boolean) returns (xr : boolean[n]; br : boolean); let
    xr, br = (xw, bw);
end

node faster(cb, cw : boolean) returns (prop : boolean); var w before \_b : boolean;
let
    w before \_b = if cw then true
        else if cb then false
        else (false \rightarrow \text{pre w before \_b});
    tells that there is an unmatched \_w
    prop = not (cw and (false \rightarrow \text{pre w before \_b}));
this node implements (77)
end

node firstafter(cb, cw : boolean) returns (cbw : boolean); var waiting boolean;
let
    cbw = cb and (false \rightarrow \text{pre waiting})
    waiting = if cw then true
        else if cb then false
        else (false \rightarrow \text{pre waiting});
this node implements (77)
end

node vec<\_w(xw : boolean[n]; xr : boolean[n]) returns (prop : boolean);
var aux : boolean[n+1];
let
    aux[0] = true;
    aux[1..n] = aux[0..n-1] and (xr = xw);
    prop = aux[\_e];
end

node compare(cw : boolean; xw : boolean[n]; xr : boolean[n]) returns (prop : boolean);
var equal : boolean; last : boolean[n] unmatched : boolean;
let
    last x if equal then xw else (init \rightarrow \text{pre last});
    stores the value to be matched
    equal = vec<\_w(xr, (init \rightarrow \text{pre last}));
    tells whether the value to be matched is actually matched
    unmatched = if cw and not (true \rightarrow \text{pre equal}) then true
        else if equal then false
        else false \rightarrow \text{pre unmatched};
    tells that there are two values waiting for match
    prop = not (cw and (false \rightarrow \text{pre unmatched}));
    a new value should not arrive while two values are waiting for match
end

node verify(cb, cr : boolean (x : boolean[n]) when cw)
returns (prop : boolean; xw, xr, xro : boolean[n]; bw, br : boolean; cro : boolean); let
    xw, bw = if cw then current writer(x)
        else (init, false) \rightarrow \text{pred}(xw, bw));
    xr, br = if cb then current buf((xw, bw) when cb)
        else (init, false) \rightarrow \text{pred}(xr, br));
    cro, xro = if cr then current reader(xr, br) when cr
        else (false, init) \rightarrow \text{pred}(cro, xro));
    prop = compare(cw, xw, xro);
assert faster(cb, cw) and faster(cr, firstafter(cb, cw));
these assertions implement (77) and (77)
assert \#(cw, cb, cr);
so as not to get bored by simultaneous clocks
end

(*
mocherotti% ler albert% do验证
Pollux Version 2.0
TRUE PROPERTY
mocherotti% )
The Signal proof

process protocol = {? boolean xw; event cw, cb, cr ! boolean xr, inv
  | [xb, bb, sbb] := bus [xw, writer(xw,cw), cb] % writer + bus %
  | [xb, br, shb] := reader (xb, bb, cr) % reader %
  | cr := (when switched(sbb)) default cr % condition (??) %
  | xok := ifo.2 (xw) % ifo.2 satisfies (??) %
  | inv := equal (xok, xr) % tests if xok=xr %
} where boolean bw, xb, bb, sbb, shb, 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, sbe)
  | [xb, bb, sbe] := buffer (xw, bw, cb)];
process reader = {? boolean xb, bb; event cr ! boolean xr, br, shb)
  | [yr, br, shb] := buffer (xb, bb, cr) % xw when switched (br) []
  | x := y when switching (br)] where boolean yr; end;
% switched(br) validates x %

process switched = {? boolean b ! boolean c)
  | [xb := b$1 init true | c := (b and not xb) or (not b and xb)]
  | where boolean xbl end; % c=true when b alternates %
process buffer = {? boolean x, b; event c ! boolean bx, bb, sbb)
  | [s, x, s] := 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 %
  | [s, x] := 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, x ! boolean inv
  | [i := (y and x) or (not y and not x) default inv
  | inv := i $1 init true
  | ]; where boolean i; end;
% tests if y=x %
process fio.2 = {? boolean x ! boolean xok)
  | [xok := shift.1(shift.1 (x)) ];
process shift.1 = {? boolean x ! boolean sx)
  | [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 %
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.