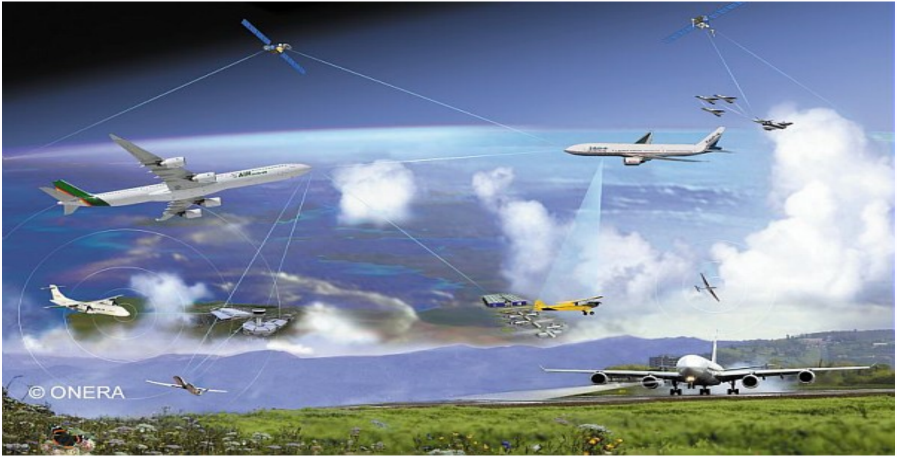


Certificate-carrying modular compilation

Guillaume Davy, Pierre-Loïc Garoche, Temesghen Kahsai, Xavier Thirioux

IRIT-CNRS, ONERA, CMU

September 2015 – GDR GPL 10emes Journées compilation

Contents

Development of critical embedded systems

- Large part of these critical systems are controllers
  - aircraft controller, engine control, medical devices, etc
- Typically designed using data-flows models
  - eg. Matlab Simulink, Scade
- Costly V&V to ensure software quality.
  - Certification regulations: DO178-C (aircraft), ISO26262 (cars), EN-50128 (railway in EU), etc
  - Process-based quality: specification of HLR (High-Level-Requirements) and (Low-Level-Requirements), traceability, conformance with standards, compliance with HLR/LLR.
  - Mainly based on test

Typical development cycle of a controller

Differential Equations (plant)

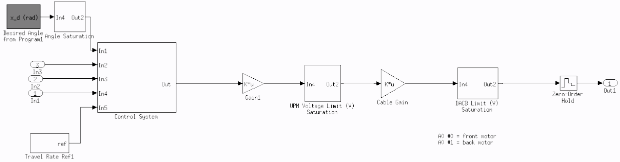
Control theorists

TYPICAL DEVELOPMENT CYCLE OF A CONTROLLER

Differential Equations (plant)

↳ Continuous controller

Control theorists



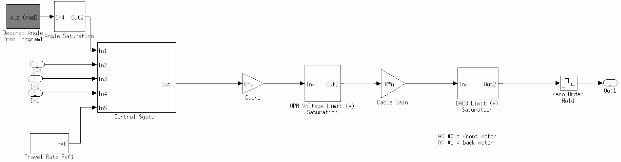
TYPICAL DEVELOPMENT CYCLE OF A CONTROLLER

Differential Equations (plant)

→ Continuous controller

→ Discrete version

Control theorists



Typical development cycle of a controller

Differential Equations (plant)

→ Continuous controller

→ Discrete version

Control theorists

Computer scientists

↓

Safety architecture

redundancy, validators,
COM/MON...

Typical development cycle of a controller

Differential Equations (plant)

→ Continuous controller

→ Discrete version

Control theorists

Computer scientists

↓

Safety architecture ⇝ Test

redundancy, validators, COM/MON...

Simulation

Typical development cycle of a controller

Differential Equations (plant)

→ Continuous controller

→ Discrete version

Control theorists

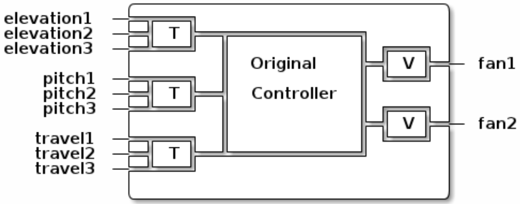
Computer scientists

↓

Safety architecture

redundancy, validators, COM/MON...

⇝ Test Simulation



Typical development cycle of a controller

Differential Equations (plant)

→ Continuous controller

→ Discrete version

Control theorists

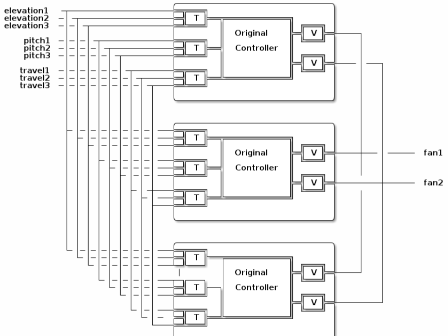
Computer scientists

↓

Safety architecture ⇝ Test

Simulation

redundancy, validators,
COM/MON...



Typical development cycle of a controller

Differential Equations (plant)

→ Continuous controller

→ Discrete version

Control theorists

Computer scientists

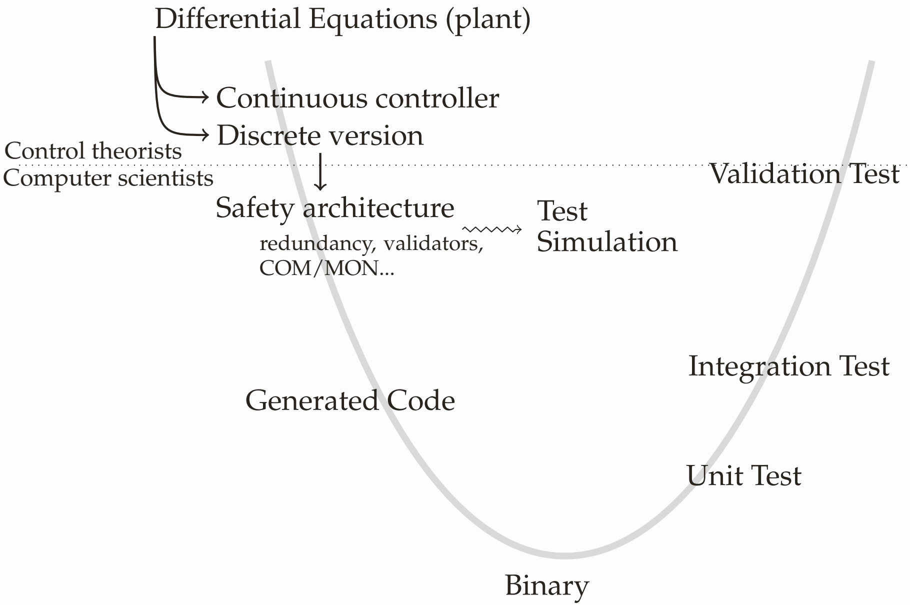
Safety architecture

redundancy, validators, COM/MON...

Test

Simulation

Typical development cycle of a controller

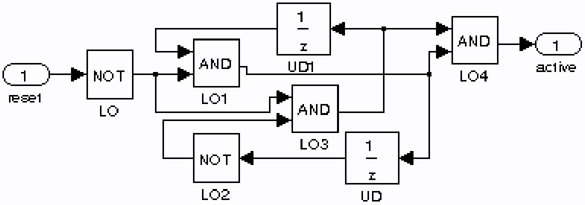


DATAFLOW MODELS: LUSTRE NODES

- Map a set of (typed) input flows to output flows.
- Not purely functional: static memory through nested `pre`

```
node counter(reset: bool) returns (active: bool);
var a, b: bool;
let
  a = false -> (not reset and not (pre b));
  b = false -> (not reset and pre a);
  active = a and b;
tel
```

- Node state characterized by its memories: `pre a` and `pre b`
- Similar construct in Matlab Simulink: Unit delay



Modular Compilation of Models

Modular compilation of synchronous languages[1]

- Node state (memories) defined by a struct

```
struct counter_mem {
 struct counter_reg { _Bool __counter_1;     // pre a
                      _Bool __counter_2;     // pre b
                    } _reg;
};
```

- One step execution by a *step* function

```
void counter_step (_Bool reset,                // input
                   _Bool (*active),            // output
                   struct counter_mem *self);// memory (side effect!)
```

- Reset function to initialize the struct

```
void counter_reset (struct counter_mem *self);
```

Open-source implementation for Lustre: LUSTRE-C

[1] D. Biernacki et al. "Clock-directed modular code generation for synchronous data-flow languages". In: *LCTES*. 2008, pp. 121–130.

Contents

Expressing the specification at model level: Synchronous observers

Requirements of our counter node:

1. output `active` is false when input `reset` holds
2. every four steps, `active` holds, starting from the 3rd one.

Synchronous observer: rely on model constructs to express the specification. Boolean output should always hold.

```
node counter_spec(reset, active: bool)
    returns (safe: bool);
var cpt: int;
let
 cpt = 0 -> if (pre cpt = 3) or reset then 0
            else pre cpt + 1;
 safe = active = (cpt = 2);
tel
```

Annotate the node with observers:

```
 --@ ensures reset => not active;
 --@ ensures counter_spec(reset, active);
 node counter(reset: bool) returns (active: bool);
```

Specification at code level: Hoare triples

Early idea from Hoare[2]:

- express imperative program intented semantics through axiomatic semantics
- use logic to formalize pre and post-conditions
- { Pre } Code { Post }

Eg. in Frama-C[3], use ANSI/ISO C Specification Language (ACSL)[4]

```
//@ requires precondition formula;
//@ ensures postcondition formula with \result;
int f (int x; int *y) {
  ...
}
```

[2] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[3] P. Cuoq et al. "Frama-C: a software analysis perspective". In: SEFM'12. Springer, 2012, pp. 233–247.

[4] P. Baudin et al. *ACSL: ANSI/ISO C Specification Language.*

Synchronous observers as Hoare triples

Simple observers (no memory) directly expressed as `ensures` statements

```
//@ ensures reset => not *active;
void counter_step (_Bool reset,
                   _Bool *active,
                   counter_mem *self) {
  ...
}
```

More complex observers may have their own memories: Stateful observers.

Contents

Stateful observers

Stateful observers are expressed as code level through:

1. observer memory, attached to the node memory definition
2. computation of the observer output using node signals *and* observer memory
3. side-effect update of the observer memory, performed at each node step execution

STATEFUL OBERVERS: EXPRESSING MEMORY

For the following contracts ,

```
--@ ensures counter_spec(reset, active);
--@ ensures reset or pre(reset) => not active
node counter(reset: bool) returns (active: bool);
```

need of additional memories:

- `pre cpt` for `counter_spec` and
- `pre reset` for `reset or pre(reset) => not active`

We enrich the node struct with additional ghost fields:

```
struct counter_mem {
  struct counter_reg {
    _Bool __counter_1;
    _Bool __counter_2;
    /*@ ghost int cpt; int cpt_s;  // pre cpt
       _Bool init1; _Bool init1_s; // initial state of cpt
       _Bool reset; _Bool reset_s; // pre reset
       _Bool init2; _Bool init2_s; // initial state of reset
    */
  } _reg;
};
```

Stateful obervers: computation of the observer property

Observer value computed on this extended memory.

ACSL expression of the Lustre node `counter_spec` semantics.

```
/*@ predicate counter_spec
  (int reset, int active, struct counter_mem *self)=
\let cond = ((self->_reg.cpt_s == 3) || reset);
\let cpt = (self->_reg.init1_s ?(0):
     ((cond?(0):((self->_reg.cpt_s + 1)))));
 (active == (cpt == 2));
*/
```

ACSL expression of the second `ensures`.

```
/*@ predicate prop
 (int reset, int active, struct counter_mem *self)=
 (self->_reg.init2_s ?(1):
 (((reset || self->_reg.reset_s) ==> (!active))));
*/
```

Only *reads* memory. No update yet.

STATEFUL OBERVERS: UPDATE OF GHOST FIELDS

End of the node step function extended to update the ghost fields:

```
void counter_step (_Bool reset, _Bool (*active),
                   struct counter_mem *self) {
 counter_reg _pre = self->_reg;
 _Bool a = _pre.__counter_2;
 _Bool b = !_pre.__counter_1;
 *active = (a && b);
 self->_reg.__counter_2 = a;
 self->_reg.__counter_1 = b;

 /*@ ghost _Bool cond; int cpt;
 cond = ((self->_reg.cpt == 3) || reset);
 if (self->_reg.init1 || cond) { cpt = 0; } else {
     cpt = (self->_reg.cpt + 1);
 }
 self->_reg.init1_s = self->_reg.init1;
 self->_reg.init1 = 0;
 ...
 self->_reg.reset_s = self->_reg.reset;
 self->_reg.reset = reset;
  */
 return;
}
```

STATEFUL OBERVERS: SUMMARY

- New memory fields:

```
      struct node_mem { struct node_reg {
          ... existing fields ...
          /*@ ghost ghost_fields */
        } _reg;
      };
```

- Predicates to denote specification

```
/*@ predicate node_spec(input, output, extended_memory) = ... */
```

- Function body: side effects in observer memories

```
void node_step (input, *output , *extended_memory) {
  ... existing code ...
  /*@ ghost ghost_fields update */
 return;
}
```

- Function contract

```
/*@ ensures node_spec(input, *output, *extended_memory); */
void node_step (input, *output , *extended_memory) { ... }
```

Contents

Verification with Frama-C

ACSL specification used to verify the code with respect to HLR

Runtime evaluation: dynamic analysis

C code instrumented to evaluate the annotations at runtime. When applied to a test bench it evaluates that all tests satisfy the property.

$\Longrightarrow$ E-ACSL plugin of Frama-C[a]

[a] Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language.*

Formal verification using weakest precondition (WP analysis)

Proofs performed at model levels using model-checking can be replayed at code/ACSL level.

k-induction[a] proofs in Lustre $\Longrightarrow$ expression as WP objectives

[a] T. Kahsai and C. Tinelli. "PKIND: A parallel *k*-induction based model checker". In: *PDMC*. vol. 72. EPTCS. 2011, pp. 55–62.

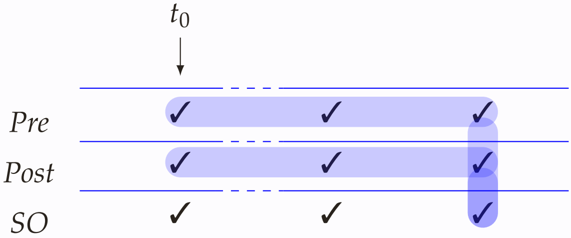
Synchronous extension of Hoare triples to flows

$$\{Pre(state, inputs)\}node(in, out)\{Post(state, state', in, out)\}$$

means

$$\square\left(\bigwedge \begin{array}{l} \mathcal{H}(Pre(state, input)) \\ node(state, state', in, out) \end{array} \implies Post(state, state', in, out)\right)$$

with $\mathcal{H}(p) \triangleq$ p has held since beginning



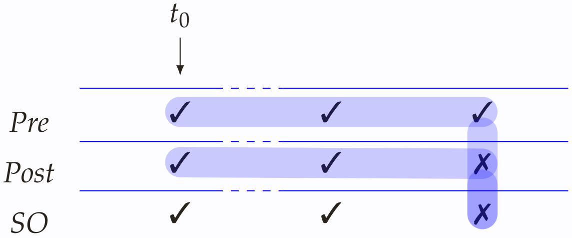
Synchronous extension of Hoare triples to flows

$$\{Pre(state, inputs)\}node(in, out)\{Post(state, state', in, out)\}$$

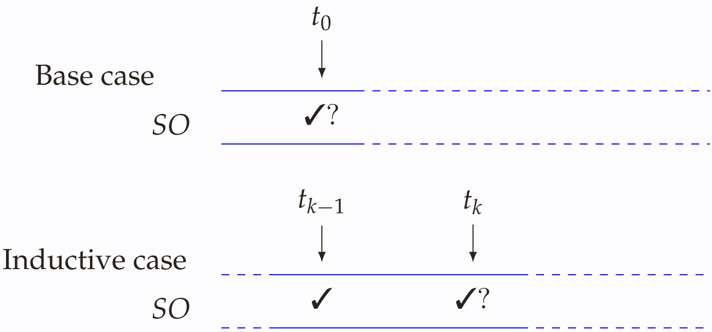
means

$$\square \left(\bigwedge \begin{array}{l} \mathcal{H}(Pre(state, input)) \\ node(state, state', in, out) \end{array} \implies Post(state, state', in, out) \right)$$

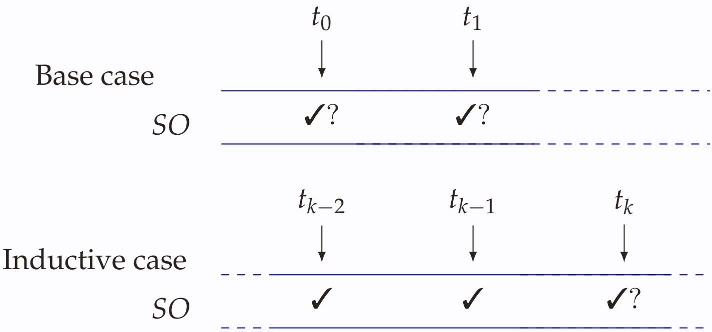
with $\mathcal{H}(p) \triangleq$ p has held since beginning



PROPERTY SO WAS PROVED INDUCTIVE



Property SO was proved k-inductive



Property SO was proved k-inductive (cont'd)

Base case

	t_0	t_1
Pre	✓	✓
Post	✓?	✓?
SO	✓?	✓?

Inductive case

	t_{k-2}	t_{k-1}	t_k
Pre	✓	✓	✓
Post	✓	✓	✓?
SO	✓	✓	✓?

Expression k-inductiveness at code level

Previous version was too naive (or good only for dynamic checking)

```
//@ ensures reset => not *active;
void counter_step (_Bool reset,
                   _Bool *active,
                   counter_mem *self) {
  ...
}
```

The property is 3-inductive:

```
//@requires Init(s) && Pre(s)
//@ensures Post(s)
```

```
//@requires \exists s1, Init(s1) && Pre(s1) && Pre(s) && Step(s1, s)
//@ensures Post(s)
```

```
//@requires \exists s1,s2, Init(s2) && Pre(s2) && Pre(s1) && Pre(s)
//@         && Step(s2,s1) && Step(s1, s)
//@ensures Post(s)
```

```
//@requires \exists s1,s2, Pre(s2) && Pre(s1) && Pre(s)
//@        && Step(s2,s1) && Step(s1, s) && Post(s2) && Post(s1)
//@ensures Post(s)
```

PLaying with proof objectives: Equivalent formulation integrating Post in BMC

Since all BMC PO should be proved, one can write them as

```
//@requires Init(s) && Pre(s)
//@ensures Post(s)
```

```
//@requires \exists s1, Init(s1) && Pre(s1) && Pre(s)
//@         && Step(s1, s) && Post(s1)
//@ensures Post(s)
```

```
//@requires \exists s1,s2, Init(s2) && Pre(s2) && Pre(s1) && Pre(s)
//@         && Step(s2,s1) && Step(s1, s) && Post(s2) && Post(s1)
//@ensures Post(s)
```

```
//@requires \exists s1,s2, Pre(s2) && Pre(s1) && Pre(s)
//@         && Step(s2,s1) && Step(s1, s) && Post(s2) && Post(s1)
//@ensures Post(s)
```

K-INDUCTION IN ONE PO

Encode multiple objectives :

$$(A_1 \implies B) \wedge (A_2 \implies B) \wedge \ldots \wedge (A_n \implies B)$$

into one

$$(A_1 \vee A_2 \vee \ldots \vee A_n) \implies B$$

Prefix definition:

$$\begin{aligned} \mathit{Prefix}_0 &= (\mathit{Post}(s) \vee \mathit{Init}(s)) \wedge \mathit{Pre}(s) \\ \mathit{Prefix}_{k+1} &= (I(s) \vee (\exists s', \mathit{Prefix}_k(s') \wedge \mathit{Step}(s', s) \wedge \mathit{Post}(s))) \wedge \mathit{Pre}(s) \end{aligned}$$

Example revisited

```
//@ Prefix3(true, reset => not *active, Step)
//@ ensures reset => not *active;
void counter_step (_Bool reset,
                   _Bool *active,
                   counter_mem *self) {
  ...
}
```

is equivalent to

```
/*@ requires (Init(self) ||
(\exists self1, reset1, active1, Init(self1)
 && Step(self1, self, reset1, active1) && reset1 => not *active1 )
||
(\exists self1, reset1, active1, self2, reset2, active2, Init(self2)
&& Step(self2, self1, reset2, active2) && reset2 => not *active2
&& Step(self1, self, reset1, active1) && reset1 => not *active1)
||
(\exists self1, reset1, active1, self2, reset2, active2,
Step(self2, self1, reset2, active2) && reset2 => not *active2
&& Step(self1, self, reset1, active1) && reset1 => not *active1)
*/
//@ ensures reset => not *active;
void counter_step (_Bool reset, _Bool *active, counter_mem *self) {...}
```

Proving optimized code

- Frama-C/WP is not able to discharge the PO
- we have to associate a predicate Init to counter_init and a Step to counter_step

The two remaining PO capture this:

(i) `//@ensures Init(mem) void N_init (mem* )`

(ii) `//@ensures Step(s1,s2, in ,out) void N_step (mem1, mem2, in , out)`

They are discharged with WP plugin.
The approach authorizes the use of code optimization:

- live variable analysis
  - minimize the memory footprint wrt a given instruction scheduling
  - maintain shared subexpressions

thanks to

- (automatic) generation of supporting ACSL annotations
  - maintaining the relationship between live variables
  - easing the automatic proof of (i) and (ii)

Verification with Frama-C - WP

For a complete analysis, additional annotations are automatically generated:

- validity of pointers
- separation of pointer aliases
- identification of modified variables (assigns)

```
/*@ requires \valid(active);
  @ requires \valid(self);
  @ requires \separated(active, &self->reg.__counter_1,
  @                     &self->reg.__counter_2);
  @ assigns *active, self->reg.__counter_1,
  @         self->reg.__counter_2;
*/
void counter_step (_Bool reset, _Bool (*active),
                   struct counter_mem *self) { ... }
```

All theses annotations are checked and support the formal analyses of encoded HLR.

Conclusion

- Context : Toolchain Simulink → Lustre → C code → executable.
- Aim : verify HLR on executable
  - Simulink → Lustre and C code → executable are assumed correct.
  - verification hard at code level, easier at model level.
  - use of formal methods to ascertain correction (no testing).
  - fully automatic.
- Proposition :
  1. express HLR at model level, as synchronous observers.
  2. check them.
  3. carry properties and proofs over to the code level.
  4. support the revalidation of properties at code level