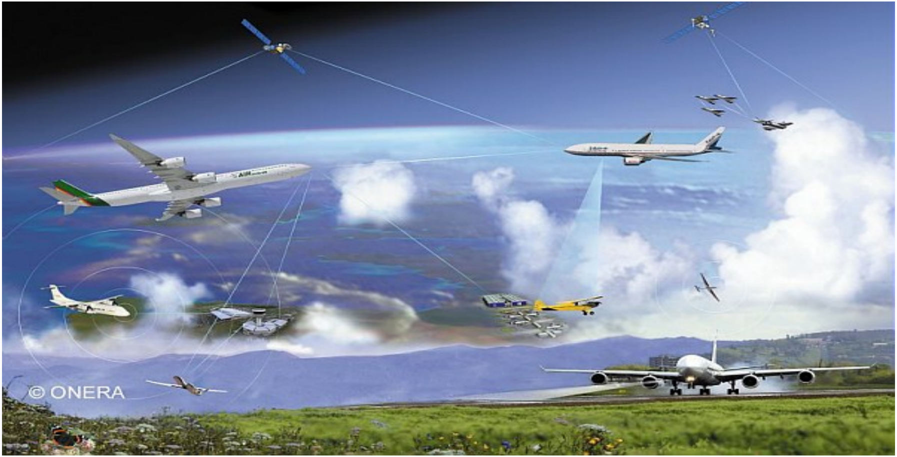


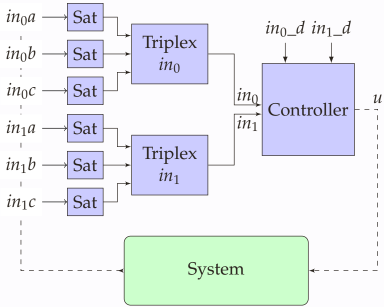
Formal verification of controller implementation our experience with Frama-C

Pierre-Loïc Garoche, with the contributions from X. Thirioux (IRIT), P. Roux (Onera), T. Kahsai (NASA/CMU), E. Feron (Georgia Tech), G. Davy (Onera), H. Herencia-Zapana (ex-NIA, now GE), R. Jobredeaux (ex-GT), T. Wang (ex-GT, now UTRC)

June 20th, 2016 – Frama-C days

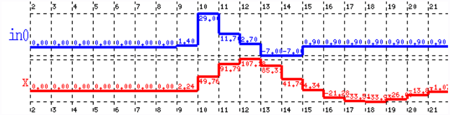
CONTEXT: CRITICAL EMBEDDED CONTROLLERS

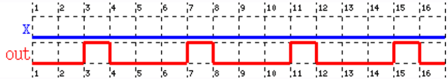
- Core elements of runtime systems
- Designed with dataflow models
  - ∗ validation through simulation/test
  - ∗ code generation
- Infinite behavior: endless loop



Designed by local composition:

- a linear controller
- combined with safety constructs





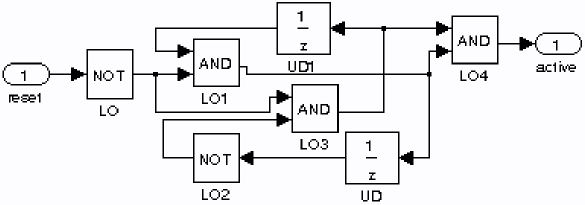
Most properties are analyzed locally.

DATAFLOW MODELS: E.G. 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



Two experiences with Frama-C

Goal: validate functional properties at code level

Two settings:

1. **linear controller: numerical core**
   * boundedness (no overflow)
   * stability, robustness (control level properties)
2. **safety constructs: voters, alarms, counters**
   * mainly integers and booleans, few serious numerical computation
   * interested in functional soundness

Global approach

- proof at model level
- automatically validation at code level
- generating ACSL during autocoding
  * for contracts: compile specification
  * for proof artifacts: compile proofs
- WP/PVS/Coq to discharge the proofs

Modular Compilation of Models[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.

Experience 1: Linear controllers

Main system

```
float in, *out;
f_mem *mem;
f_reset(mem):
while (true) {
  in = receive_input();
  f_step(in, out, mem);
  send_output(*out);
}
```

- Boundedness: loop invariant eg. I(in, *out, mem) or I(mem)
- As function contracts

```
/*@ ensures I(mem); */
void f_reset (...);

/*@ requires I(mem);
  @ ensures I(mem); */
void f_step (...);
```

Need for super-linear invariants

Let A be a square matrix. Define the linear system:

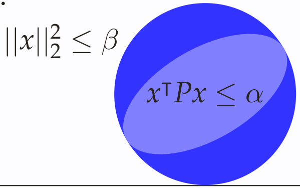
$$x^{k+1} = Ax^k, k \geq 0, \text{ a given } x^0$$

A matrix P satisfies Lyapunov conditions for the system iff:

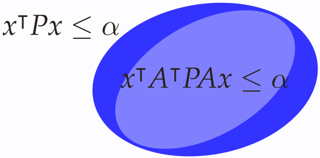
$$P - \mathrm{Id} \succeq 0 \,, \qquad P - A^\intercal P A \succ 0$$

- Id is the identity matrix;
- $M \succ 0$ means $M = M^\intercal$ and $\forall\, x \neq 0, x^\intercal M x > 0$;
- $M \succeq 0$ means $M = M^\intercal$ and $\forall\, x, x^\intercal M x \geq 0$.

$P - \mathrm{Id} \succeq 0$ implies boundedness:



$P - A^\intercal P A \succ 0$ guarantees the strict decrease:



Need for super-linear invariants

Let A be a square matrix. Define the linear system:

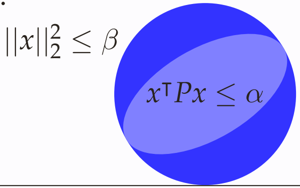
$$x^{k+1} = Ax^k, k \geq 0, \text{ a given } x^0$$

A matrix P satisfies Lyapunov conditions for the system iff:

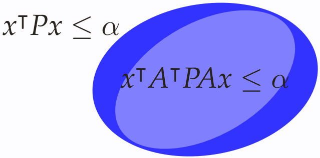
$$P - \mathrm{Id} \succeq 0 \,, \qquad P - A^\intercal P A \succ 0$$

- Id is the identity matrix;
- $M \succ 0$ means $M = M^\intercal$ and $\forall\, x \neq 0, x^\intercal M x > 0$;
- $M \succeq 0$ means $M = M^\intercal$ and $\forall\, x, x^\intercal M x \geq 0$.

$P - \mathrm{Id} \succeq 0$ implies boundedness:



$P - A^\intercal P A \succ 0$ guarantees the strict decrease:



Need for quadratic invariant (at least)!

Step 1: teach linear algebra to Frama-C

```
/*@ axiomatic matrix {
 type LMat;
 ...
 logic LMat transpose(LMat x0);
 logic real dot(LMat x0, LMat x1);
 logic LMat diag(LMat x0);
 logic LMat inv(LMat x0);
 ...
 logic real dot_inner(LMat x0, LMat x1, integer x2) =
  (((x2==(-1)))?((0.0)):(((mat_get(x0, x2, (0))*mat_get(x1,
     x2, (0)))+dot_inner(x0, x1, (x2-(1))))));
 axiom dot_def:
   (\forall LMat A; (
    (\forall LMat B; (((getM(A)==getN(A))==>(dot(A, B)
       ==dot_inner(A, B, (getM(A)-(1))))))
    ))
   ));
 logic in_ellipsoid LMat (LMat P, LMat x);
 axiom in_ellipsoid_def: ...
```

Step 2: Generation of function contract / loop invariant

Using convex optimization tools, we synthesize the discrete Lyapunov function at model level: $P = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$

And express it at code level:

```
#define P MatCst_2_2((a), (b), (c), (d))

struct mem { double x1, x2;    /* pre x1, pre x2 */ };

/*@ requires in_ellipsoid(P, VectVar_2(mem->x1, mem->x2));
  @ ensures  in_ellipsoid(P, VectVar_2(mem->x1, mem->x2)); */
void linctl (in, *out, mem);
```

SMT solvers behind Why3 (z3, yices, alt-ergo, cvc4) do not succeed.
Our solution:

- generate simple intermediate proof objectives
  - thanks to generation of local invariants
- prove them with a proof assistant (PVS)

STEP 3: LOCAL REASONING

Ellipsoids propagation in linear code: two mains theorems

1. linear transformation of an ellipsoid
   We define ξ_P as $\{x \mid x^\intercal P x \leq 1\}$

$$x \in \xi_P \wedge y = Ax \implies y \in \xi_{APA^\intercal}$$

2. combination of two ellipsoids (S-procedure)

$$\exists \tau_1, \tau_2 \in \mathbb{R}^+, \begin{bmatrix} -P & 0 \\ 0 & 1 \end{bmatrix} - \tau_1 \begin{bmatrix} -P_1 & 0 \\ 0 & 1 \end{bmatrix} - \tau_2 \begin{bmatrix} -P_2 & 0 \\ 0 & 1 \end{bmatrix} \succeq 0$$

is a sufficient condition for

$$\left(x^T P_1\, x \leq 1 \wedge x^T P_2\, x \leq 1 \right) \Rightarrow x^T P\, x \leq 1$$

Used to generate local assert, propagating loop invariant:

```
/*@ requires in_ellipsoid(P,VectVar_2(mem->x1, mem->x2));
  @ ensures in_ellipsoid(Q,VectVat_3(mem->x1, mem->x2,v));*/
  @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
{
// assignment of v
}
```

STEP 4: TEACH LINEAR ALGEBRA TO PVS

Development of an ellipsoid library in PVS

- basic datatypes: vector, matrices, ...
- specific predicates: `in_ellipsoid`
- two main theorems

```
ellipsoid_general: THEOREM
 forall (n:posnat,m:posnat, Q:SquareMat(n),
        M: Mat(m,n), x:Vector[n], y:Vector[m]):
  in_ellipsoid_Q?(n,Q,x) AND y = M*x
  IMPLIES
  in_ellipsoid_Q?(m,M*Q*transpose(M),y)

ellipsoid_combination: THEOREM ..
```

Local contract are proved thanks to

- computation of proof objective by Frama-C
- appropriate choice of proof strategies generated when propagating ellipsoids.

Automatic framework

- Compute the Lyapunov function: (semi-)automatic
- In the code generator
  - embedded C code
  - function contracts with loop invariant
  - statement-local annotations with
    - propagated ellipsoid
    - proof strategy annotation
- In Frama-C
  - a plugin that
    - declare the grammar extension
    - calls WP/Why3/PVS with the appropriate strategy

Implemented in Geneauto+ by Tim Wang and Romain Jobredeaux.

LINEAR CONTROLLER ANALYSIS – SUMMARY

1. computation of a Lyapunov function, a quadratic invariant
2. compilation of the invariant along the code as ACSL contracts
3. **inductiveness** proof on the code using Frama-C WP/Why /PVS

We also check externally (outside Frama-C) that the floating point errors generated in one loop iteration do not break inductiveness.

$$A^t PA - P + |noise| \preceq 0$$

Current extensions include

- analysis at model and code level of closed loop properties
  * express the plant semantics in ACSL ghost code
  * robustness through vector margins
  * performance properties, eg. overshot, related to output H_∞ norm
- more sophisticated systems and properties thanks to the integration of SOS programming in Alt-Ergo.
  $\Rightarrow$ do not require PVS and the statement level annotations.

Experience 2: Safety constructs

Node semantics expressed by another node: a **synchronous observer**

```
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);
```

SMT-based model-checking proves these properties invariant:

- `node` and `spec` expressed as SMT predicates: $I(s)$, $T(s, s')$ and $P(s)$
- Induction proof: $I(s) \models P(s)$ and $P(s) \wedge T(s, s') \models P(s')$

Issue #1: express semantics at code level

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.

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`

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 as ACSL predicates

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 semantics: update of ghost fields

Ghost code to update 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, ext_memory) = ... */
```

- Function body: side effects in observer memories

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

- Function contract

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

Issue #2: Verification with Frama-C

ACSL used to verify the code with respect to specification

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.

Proving code, attach ACSL semantics to code

- Frama-C/WP is not able to discharge the PO:
  1. P is not inductive over T
     (eg. k-induction, or need of additional invariants)
  2. function `N_step` was optimized or too complex

Proving code, attach ACSL semantics to code

- Frama-C/WP is not able to discharge the PO:
  1. P is not inductive over T
     (eg. k-induction, or need of additional invariants)
  2. function `N_step` was optimized or too complex
- Solution #2: we generate ACSL encoding of function semantics

  predicates `Init` for `counter_init` and
  `Step` for `counter_step`

We define the two additional ensure statements:

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

(ii)
```
/*@ensures Step(s1,s2, in ,out)
   ensures node_spec(input, *output, *ext_memory); */
void N_step (mem1, mem2, in , out)
```

Proving code, attach ACSL semantics to code

- Frama-C/WP is not able to discharge the PO:
  1. P is not inductive over T
     (eg. k-induction, or need of additional invariants)
  2. function `N_step` was optimized or too complex
- Solution #2: we generate ACSL encoding of function semantics
  predicates `Init` for `counter_init` and
  `Step` for `counter_step`

We define the two additional ensure statements:

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

(ii)
```
/*@ensures Step(s1,s2, in ,out)
   ensures node_spec(input, *output, *ext_memory); */
void N_step (mem1, mem2, in , out)
```

Two main proof objectives:

1. Prove `node_spec` wrt `Init` and `Spec`
   Was done with similar predicates at model level with the same SMT solvers
2. Prove that `N_step` refines `Spec`

Optimized code and refinement proof

In case of optimized code, difficulties to prove

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

Eg. limit the number of stack allocation through variable reuse
⇒ makes the WP relationship less tractable.

- variable liveness analysis
  - minimize the memory footprint wrt a given instruction scheduling
  - maintain shared sub-expressions

Additional statement local asserts are introduced to keep track of relationships

- (automatic) generation of supporting ACSL annotations
  - introduce simpler pointer-less struct
  - maintain relationship between live variables
  - ease the automatic proof of (i) and (ii)

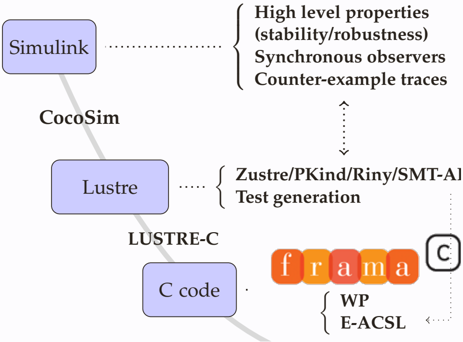
Safety constructs analysis – summary

1. compilation of specifications (synchronous observers) as
   * ACSL predicates
   * ghost fields (stateful observers)
   * ghost code (side effects on observers memory)
2. compilation of models as ACSL predicates
3. additional statement level annotations for optimized code
4. proof with Frama-C/WP of
   * (k-)inductiveness on model and specification ACSL predicates
   * refinement between code and ACSL predicates

Current extensions include

- complete implementation of the approach
- extension to stateflow (hierarchical states automata)
- adapt the proof strategy at code level to the ones performed at model level
  * PDR proof as induction proof
  * k-induction
  * export of additional invariants

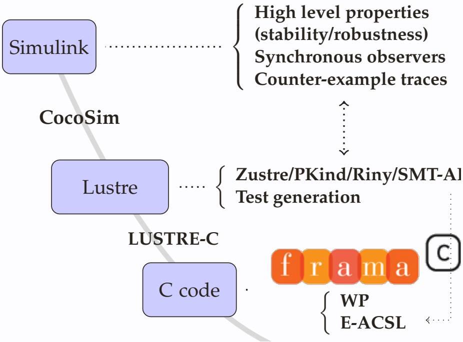
VISION: INTEGRATE FORMAL METHODS IN THE DEV. CYCLE



Large use of Frama-C at C level:

- generation of ACSL (predicates, axioms)
- development of plugins
- grammar extensions
- proof strategies for PVS

VISION: INTEGRATE FORMAL METHODS IN THE DEV. CYCLE



Large use of Frama-C at C level:

- generation of ACSL (predicates, axioms)
- development of plugins
- grammar extensions
- proof strategies for PVS

Thank you.
Any question?

OPTIMIZED CODE 1/2

Pointer-less structs

```
/* Struct definitions */
struct f_mem {struct f_reg {int __f_2; } _reg; struct
    _arrow_mem *ni_2; };
//@ ghost struct f_mem_pack {struct f_reg _reg; struct
    _arrow_mem_pack ni_2;
};
```

Optimized code 2/2

Keeping track of live variables

```
//@ assert \forall struct f_mem_pack mem1; \forall struct
    f_mem_pack mem2; \at(f_pack2(mem1, self), Pre) ==> f_pack0
    (mem2, self) ==> trans_fA(x, mem1, mem2, *y);
*y = (x + 1);
//@ assert \forall struct f_mem_pack mem1; \forall struct
    f_mem_pack mem2; \at(f_pack2(mem1, self), Pre) ==> f_pack0
    (mem2, self) ==> trans_fB(x, mem1, mem2, *y);
```

with

```
/*@ predicate trans_fy(int x_in, struct f_mem_pack mem_in,
    struct f_mem_pack mem_out, int y_out) = (y_out == x_in +
    1);*/
 /*@ predicate trans_fB(int x_in, struct f_mem_pack mem_in,
     struct f_mem_pack mem_out, int y_out) = trans_fA(x_in,
     mem_in, mem_out, y_out) && (clock_fy(x_in, mem_in,
     mem_out) ==> trans_fy(x_in, mem_in, mem_out, y_out));
 */
```