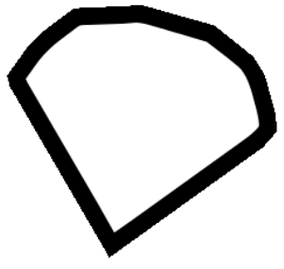
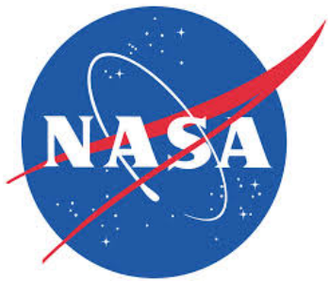


CristalCaveGem

A tool for verification and code generation
for critical systems

Temesghen Kahsai
NASA Ames / CMU



CristalCaveGem

- Work in progress
- Open source tool for formal verification and code generation
- In this talk I will share the overall objectives and initial results

Joint work with:

- Pierre-Loic Garoche (Onera - France)
- Xavier Thirioux (INPT-ENSEEIHT and Onera - France)
- Arnaud Dieumegard (INPT-ENSEEIHT - France)
- Arnaud Venet (NASA Ames / CMU - USA)

Contents

- Overall contributions and objectives
- The toolchain
- Compilation and formal verification
  - ➡ Compiling specification
  - ➡ Logic based model checking
- Use cases

Overall contributions and objectives

Focus on:

- model-based development and auto-coding
- control software
- formal specification
- formal verification

Focus on:

- model-based development and auto-coding
  - ✤ Simulink® / Lustre, compilation toolchain to C code
- control software
- formal specification
- formal verification

Focus on:

- model-based development and auto-coding
  - ✤ Simulink® / Lustre, compilation toolchain to C code
- control software
  - ✤ aircraft controllers, FADEC, safety architecture ...
- formal specification
- formal verification

Focus on:

- model-based development and auto-coding
  - ✤ Simulink® / Lustre, compilation toolchain to C code
- control software
  - ✤ aircraft controllers, FADEC, safety architecture ...
- formal specification
  - ✤ using Lustre contracts and Simulink® annotation blocks
  - ✤ automatic generation of ACSL contracts
- formal verification

Focus on:

- model-based development and auto-coding
  - ✤ Simulink® / Lustre, compilation toolchain to C code
- control software
  - ✤ aircraft controllers, FADEC, safety architecture ...
- formal specification
  - ✤ using Lustre contracts and Simulink® annotation blocks
  - ✤ automatic generation of ACSL contracts
- formal verification
  - ✤ model checking at Lustre level (i.e. safety properties)
  - ✤ Frama-C and IKOS at code level (i.e. pointer analysis)

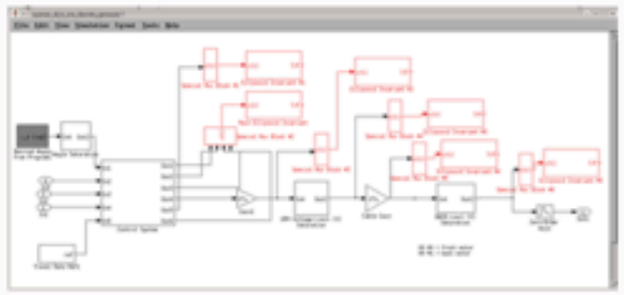
The toolchain

THE TOOLCHAIN

CristalCaveGem

Critical Systems Toolchain And Language for Code Generation And Verification, Generic and Modular

Simulink®



GENE-AUTO

Kind ..... Synchronous language (Lustre)

```
/*@ requires ∃ bi:real; s1, s2, s3 ≤ bi;
    ensures ∀ bi:real; s1, s2, s3 ≤ bi
      ⇒ sensor ≤ 2*bi; */
node voter (s1, s2, s3: real)
  returns (sensor: real);
let
...
tel
```

LUSTRE-C

frama C ..... C code

GENEAUTO

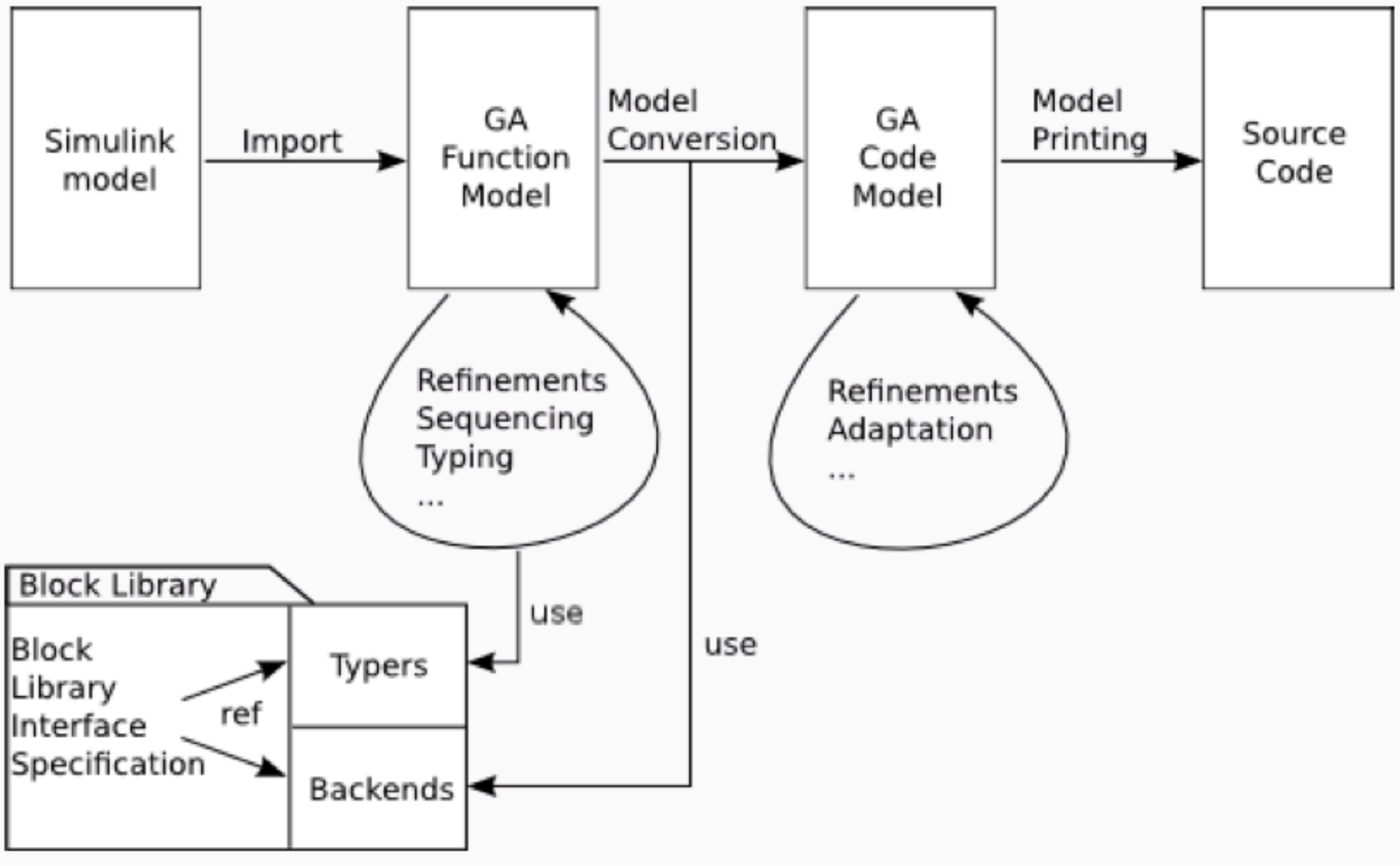
Geneauto: open-source code generation toolset (ITEA project)
`www.geneauto.org`

- input: Simulink, Stateflow or Scicos
- output: C or Ada

GENEAUTO

Geneauto: open-source code generation toolset (ITEA project)
`www.geneauto.org`

- input: Simulink, Stateflow or Scicos
- output: C or Ada



GENEAUTO

Geneauto: open-source code generation toolset (ITEA project)
`www.geneauto.org`

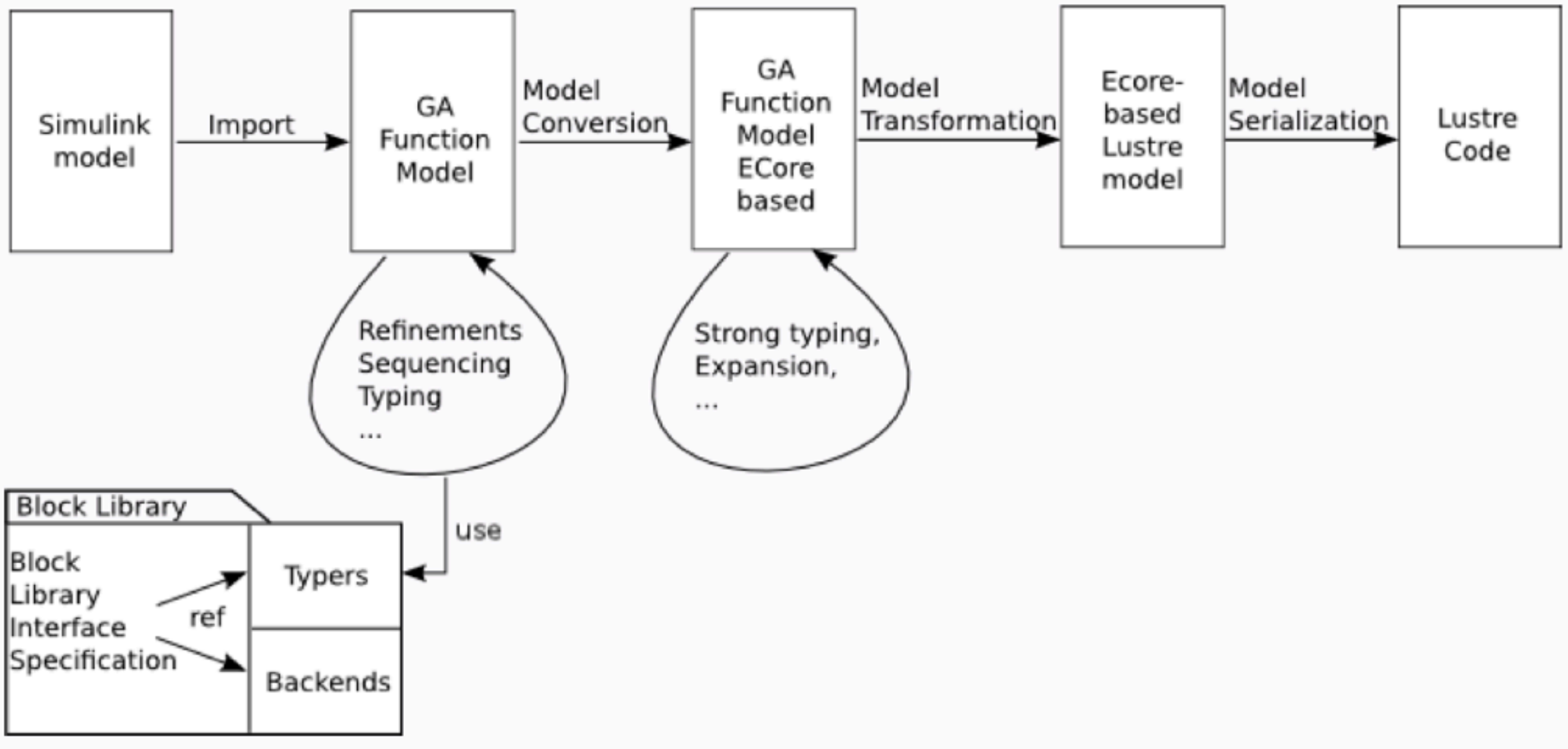
- input: Simulink, Stateflow or Scicos
- output: C or Ada

Limitations:

- Matlab Simulink r2008b (.mdl not .slx)
- DISCRETE simulink
- Not fancy blocks, no complex z-expressions, just unit delays

GENEAUTO-L

- Skip the CodeModel phase
- Map SystemModel elements to Lustre elements
- Use a textual/graphical editor for Lustre with Xtext grammar.



Compilation and formal verification

COMPILING SPECIFICATION

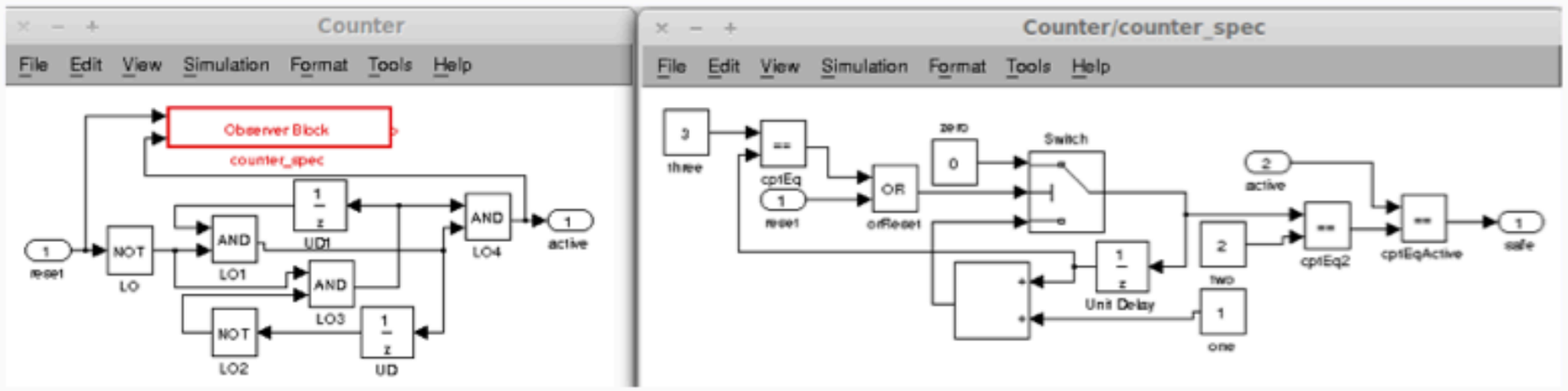
Idealy formal specification is available for each node of the system:

- traceability of requirements
- used to drive the verification process

* test with human oracle
* code review
* **formal verification**

Requirements: provide means to express specification at model level:

- Simulink annotation blocks



- Lustre contracts

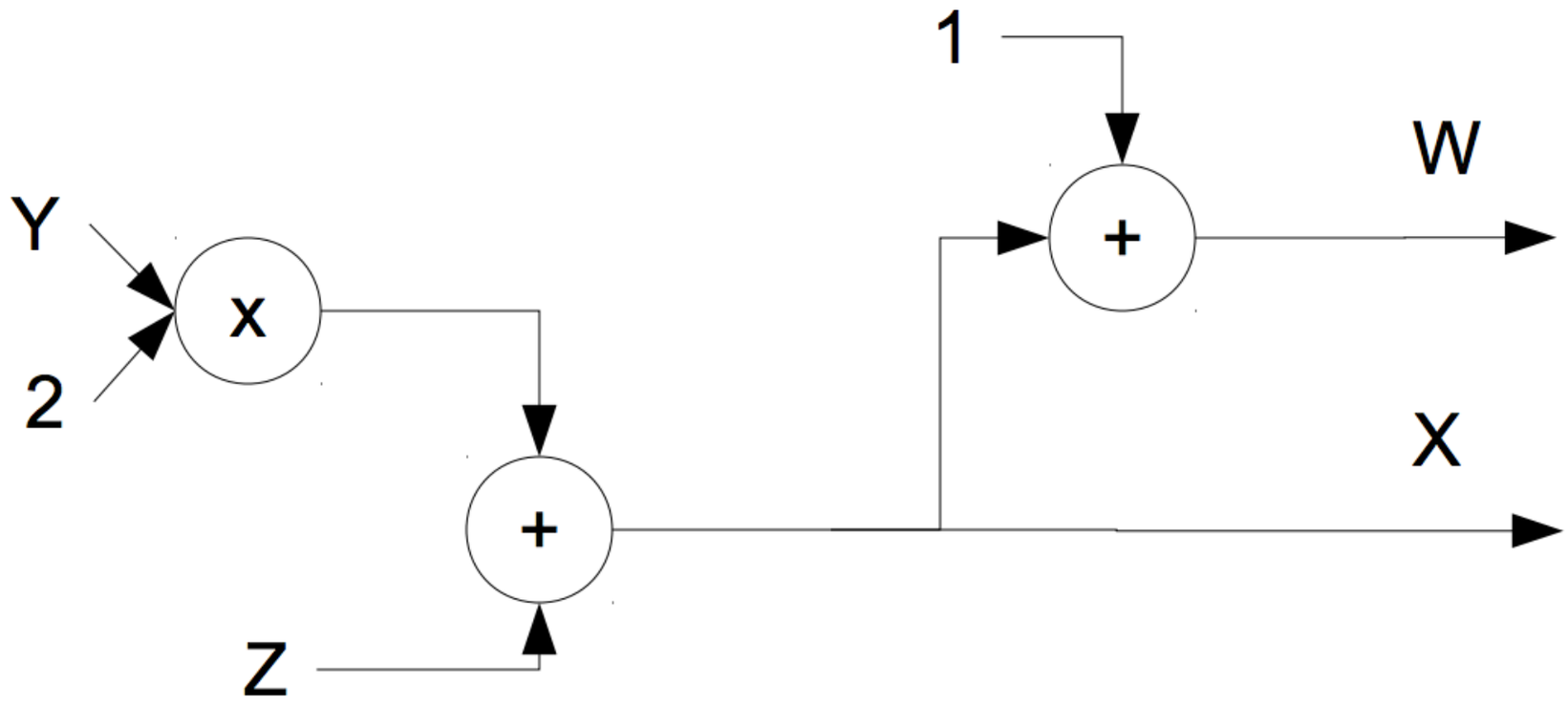
LUSTRE

- Safety-critical software must be deterministic
  - System execution behaves as expected (specs.)
- Reaction to events and associated values
  - Sensors, actuators ...
- Domain-specific and math approach
- Reaction to events and associated values
  - System specified engineers (not software engineers).

LUSTRE – DATAFLOW

- Consider data dependencies
- New data produced at each instant

$$X(t) = 2 * Y(t) + Z(t)$$
$$W(t) = X(t) + 1$$



LUSTRE

- Synchronous dataflow language.
- Declarative and deterministic specification/programming language for model-based development
- Used to model control systems
- Lustre programs = systems of equational constraints between input and output streams

LUSTRE

- Synchronous dataflow language.
- Declarative and deterministic specification/programming language for model-based development
- Used to model control systems
- Lustre programs = systems of equational constraints between input and output streams

```
node therm_control (actual: real; up, dn: bool )
          returns (heat, cool : bool)
 var desired, margin : real;
 let
    margin = 1.5;
    desired = 21.0 → if dn then (pre desired) − 1.0
                     else if up then (pre desired) + 1.0
                     else (pre desired);
    cool = (actual − desired) > margin;
    heat = (actual − desired) < −margin;
 tel
```

LUSTRE CONTRACTS

- Synchronous observers
- Expressed in Lustre but out of Lustre model – as node annotations
  - allow to separate code from specification when compiling
- Only on input and output flows – compatible with requirements definition

LUSTRE CONTRACTS

- Synchronous observers
- Expressed in Lustre but out of Lustre model – as node annotations
  - allow to separate code from specification when compiling
- Only on input and output flows – compatible with requirements definition

```
--@ requires expr1(reset);
--@ ensures expr2(reset,bool);
node counter1(reset: bool) returns (active: bool);

--@ observer counter_simplespec(reset, active);
node counter2(reset: bool) returns (active: bool);
```

SYNCHRONOUS OBSERVER

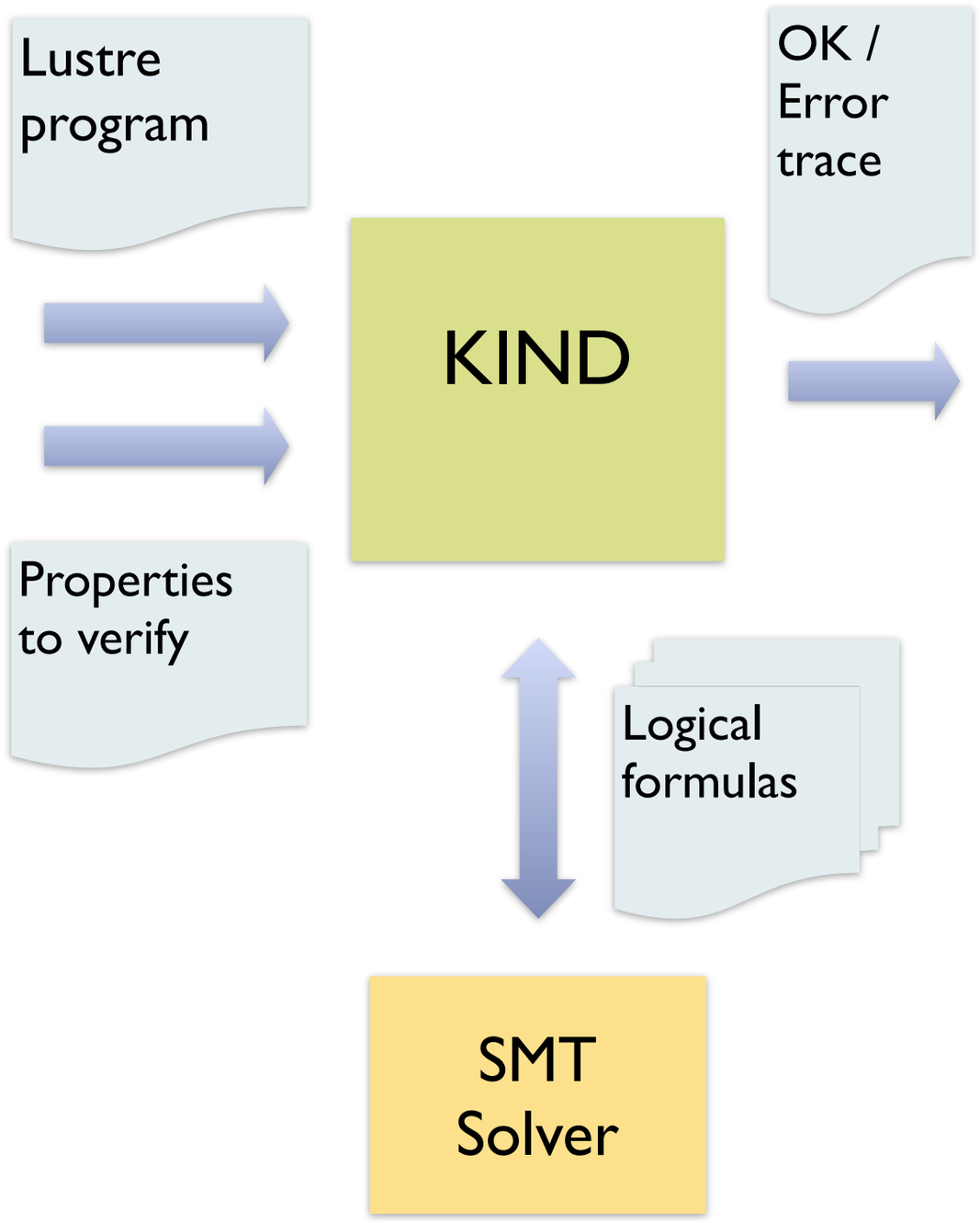
- For safety properties, instead of writing the specification as a temporal logic formula and translating it to state machine
- Just write the specification directly as a state machine
- Specifically, a state machine that is synchronously composed with the system state machine
- And that observes its state variables
- And either signals an alarm if the intended behavior is violated, or OK as long as it is not
- This is called a synchronous observer
- Then we check that alarm or not OK are unreachable

* check: *FORMULA (system || observer) |- G(NOT alarm)*

Kind model checker

- Translation of
  - a Lustre program L and
  - a putative invariant property P

  into set F of SMT formulas
- SMT-based k-induction on F to prove or disprove P for L
- Some features:
  - Parallel architecture
  - Automated invariant generation
  - Path compression
  - Abstraction/refinement ...
  - Incremental multi-property verification



LustreC

- Compiling Lustre programs in C/Java
- Open source implementation of [LCTES'08].

  Available at
  `cavale.enseeiht.fr/redmine/projects/lustrec`.
  Generate modular C/Java code: one object per Lustre node.

LustreC

- Compiling Lustre programs in C/Java
- Open source implementation of [LCTES'08].

  Available at
  `cavale.enseeiht.fr/redmine/projects/lustrec`.
  Generate modular C/Java code: one object per Lustre node.

  In C:

- one struct per node describing internal memories and states (struct) of subnodes
- one step function for a single transition

  Less efficient than other compilation scheme but ...

- used in industry (*Esterel Scade*® certified compiler *Scade*®)
- easier to argue about traceability (and verification)

COMPILING SIMPLE CONTRACTS

- Observers, ie. nodes are translated as ACSL predicates
- ACSL : ANSI/ISO C Specification Language (`http://frama-c.com/acsl.html`).

COMPILING SIMPLE CONTRACTS

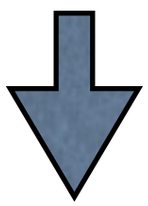
- Observers, ie. nodes are translated as ACSL predicates
- ACSL : ANSI/ISO C Specification Language (`http://frama-c.com/acsl.html`).

```
node couter_simplespec (reset, active: bool)
                  returns (safe: bool);
let
  safe = reset => not active;
tel

--@ observer counter_simplespec(reset, active);
node counter(reset: bool) returns (active: bool);
...
```

COMPILING SIMPLE CONTRACTS

- Observers, ie. nodes are translated as ACSL predicates
- ACSL : ANSI/ISO C Specification Language (`http://frama-c.com/acsl.html`).

```
node couter_simplespec (reset, active: bool)
                  returns (safe: bool);
let
  safe = reset => not active;
tel

--@ observer counter_simplespec(reset, active);
node counter(reset: bool) returns (active: bool);
...
```



LUSTREC

```
/*@ predicate counter_spec(bool reset, bool active) =
    \let safe = reset ==> not active;
    safe;
*/

//@ ensures counter_spec(reset, *active);
counter_step (_Bool reset, _Bool *active, counter_mem *self)
{ ... }
```

MEMORY IN CONTRACTS 1/2

When observers contain "pre", one needs to keep this information and perform updates.

```
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
```

Enrich memory state with annotation related fields.

```
struct counter_mem {
  struct counter_reg { _Bool __counter_1; _Bool __counter_2;
                /*@ ghost struct obs2_reg { int __pre_mem1;
                                            int __mem1;
                                          } _spec1; */
                  } _reg;
};
```

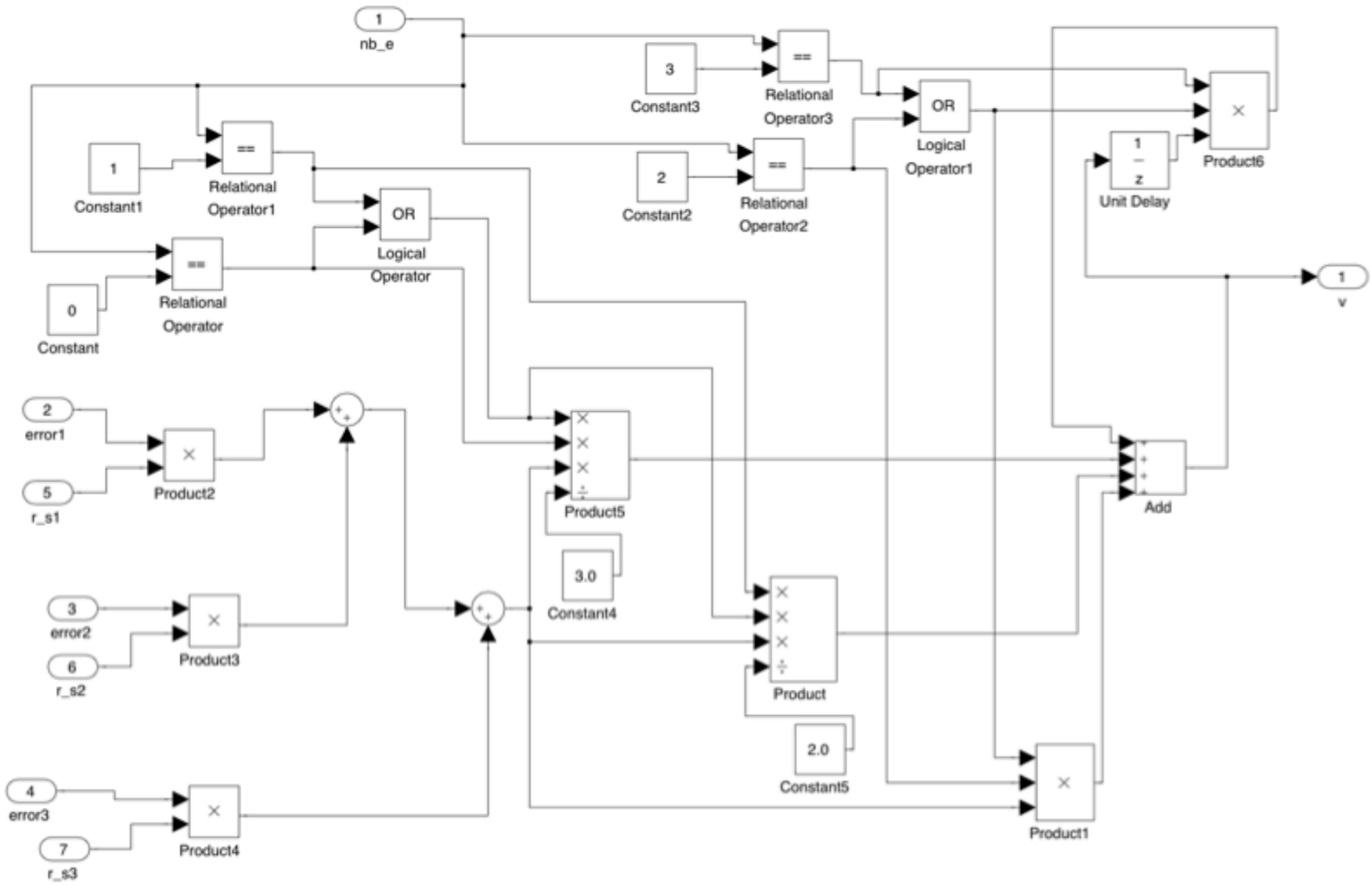
MEMORY IN CONTRACTS 2/2

```
/*@ logic bool counter_spec(bool reset, bool active, int mem1) =
active == (mem1 == 2); */
//@ ensures counter_step(reset,*active,self->_reg._spec1.__mem1);
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 int _counter_spec_3;
  _Bool __counter_spec_2 = ((_pre._spec1.__mem1 == 3) || reset);
  if (__counter_spec_2) {
    __counter_spec_3 = 0;
  } else {
    __counter_spec_3 = (_pre._spec1.__mem1 + 1);
  }
  self->_reg._spec1.__mem1 = __counter_spec_3;
  */
  return; }
```

Use cases

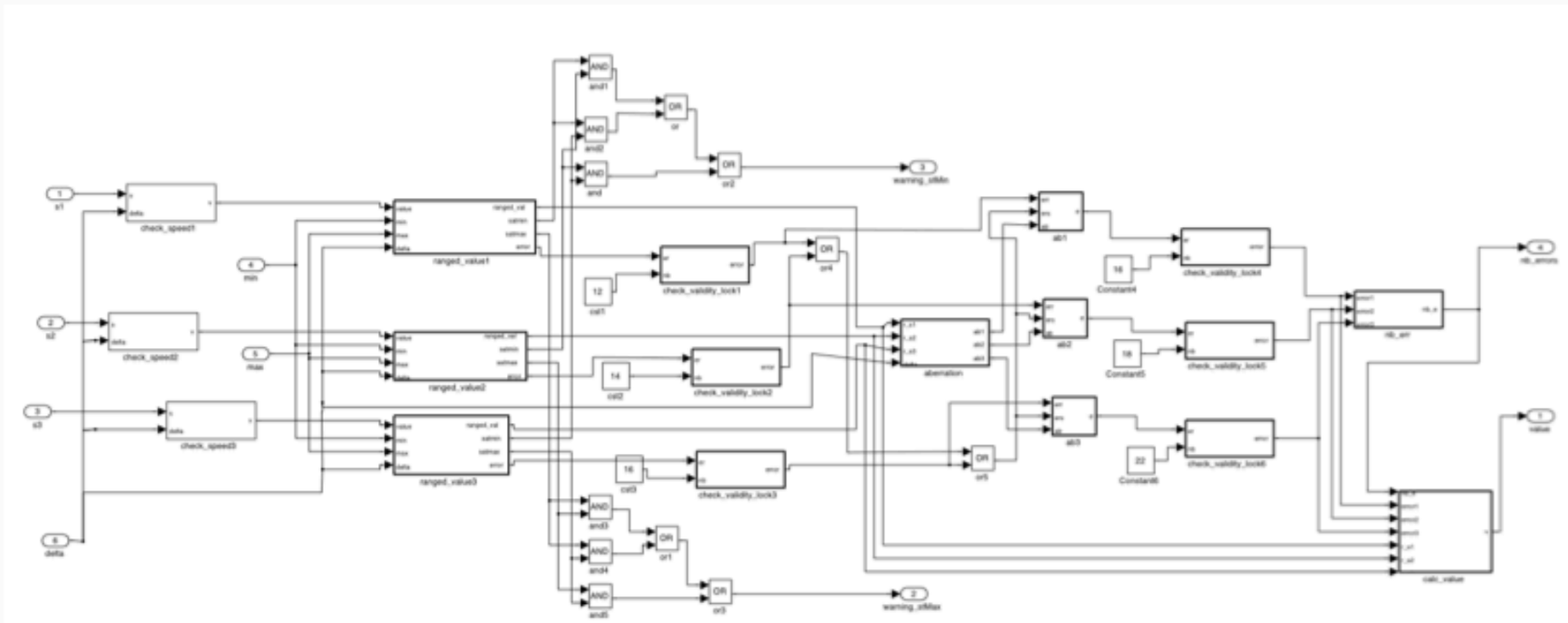
USE CASES: SAFETY ARCHITECTURE (1/6)

Developped by Alice Tailliar (ENSEEIHT).
Triplex based architecture with error management
Compute the mean value of valid inputs



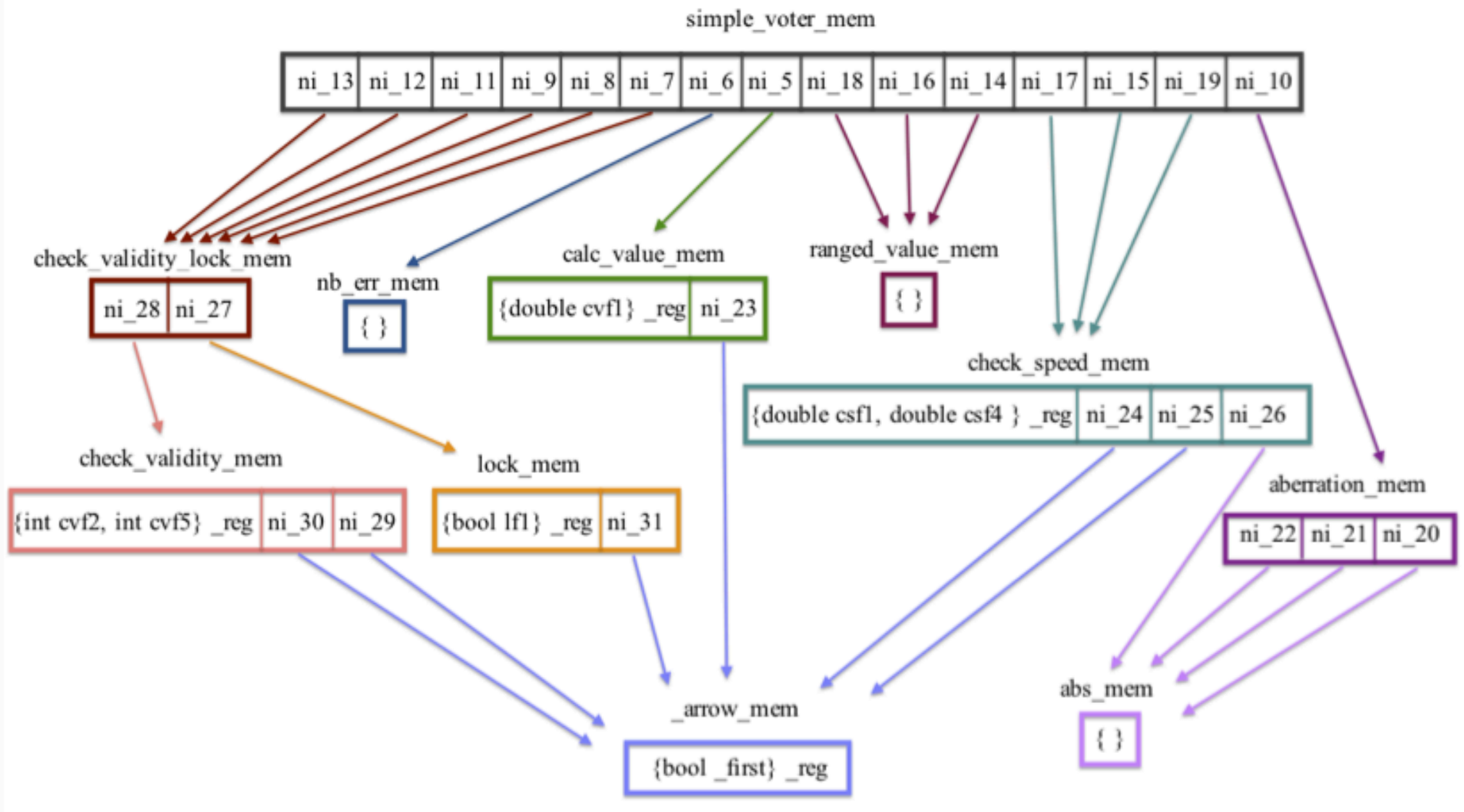
USE CASES: SAFETY ARCHITECTURE (2/6)

The voter with the error detection system



USE CASES: SAFETY ARCHITECTURE (3/6)

The complete system contains 34 nodes and 476 block instances. The generated code for the main node is associated to a complex memory (state representation).



USE CASES: SAFETY ARCHITECTURE (4/6)

```
--@ requires min <= max;
--@ requires delta > 0.0;

--@ ensures min <= value and value <= max;
--@ ensures 0 <= nb_errors and nb_errors <= 3;
--@ ensures nb_errors = 3 => value = pre value;
--@ observer obs_simple_voter(s1, s2, s3, min, max, delta, value
-- warning_stMax, warning_stMin, ok_stMin, ok_stMax,
-- ok_revMin, ok_revMax);

node simple_voter (s1,s2,s3,min,max,delta: real)
 returns (value: real; warning_stMax,warning_stMin: bool;
   nb_errors: int);
```

All nodes are annotated with Lustre contracts.

```
--@ requires nb_e = 0 or nb_e = 1 or nb_e = 2 or nb_e = 3;
--@ requires nb_e = nb_err(e1, e2, e3);
--@ ensures nb_e = 2 => (v = s1 or v = s2 or v = s3);
--@ ensures nb_e = 3 => (v = pre v);
--@ ensures (nb_e = 1 and e1) => v = (s2 + s3) / 2.0;
--@ ensures (nb_e = 1 and e2) => v = (s1 + s3) / 2.0;
--@ ensures (nb_e = 1 and e3) => v = (s1 + s2) / 2.0;
--@ ensures nb_er = 0 => v = (s1 + s2 + s3) / 3.0;

node calc_value (nb_e: int; e1,e2,e3: bool; s1,s2,s3: real)
  returns (v: real);
```

USE CASES: SAFETY ARCHITECTURE (5/6)

```
node obs_simple_voter(s1, s2, s3, min, max, delta, value : real;
                      warning_stMax, warning_stMin : bool)
  returns (ok_stMin, ok_stMax, ok_revMin, ok_revMax : bool);
var
  stmin1, stmin2, stmin3 : bool;
  stmax1, stmax2, stmax3 : bool;
let
  stmin1 = s1 < min - delta;
  stmin2 = s2 < min - delta;
  stmin3 = s3 < min - delta;
  stmax1 = s1 > max + delta;
  stmax2 = s2 > max + delta;
  stmax3 = s3 > max + delta;
  ok_stMin = (warning_stMin => ((stmin1 and stmin2)
                 or (stmin2 and stmin3) or (stmin1 and stmin3)));
  ok_stMax = (warning_stMax => ((stmax1 and stmax2)
                 or (stmax2 and stmax3) or (stmax1 and stmax3)));
  ok_revMin = (((stmin1 and stmin2) or (stmin2 and stmin3)
                 or (stmin1 and stmin3)) => warning_stMin);
  ok_revMax = (((stmax1 and stmax2) or (stmax2 and stmax3)
                 or (stmax1 and stmax3)) => warning_stMax);
tel
```

Use cases: Safety architecture (6/6)

- Kind

* All contracts proved by Kind. Each prop in 1/5 s.
* Each of the node and its subnodes are expanded and the total system analyzed.
* Some of them (4/34) need additional invariants to be k-inductive.

- Frama-C/WP

* Partial revalidation of generated contracts.
* Seem to hardly scale for the topmost nodes.
* Total of 450 Frama-C goals.
* +17k goals when using the split option.

USE CASES: NASA DOCKING STATION

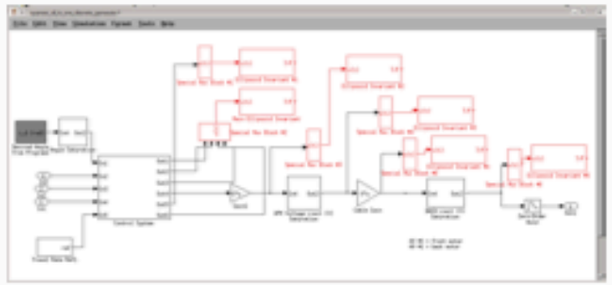
Full specification in Simulink/Stateflow and Lustre of the docking station system of the ISS.

Summary and future work

CristalCaveGem

Critical Systems Toolchain And Language for Code Generation And Verification, Generic and Modular

Simulink®



GENE-AUTO

Kind ..... Synchronous language (Lustre)

```
/*@ requires ∃ bi:real; s1, s2, s3 ≤ bi;
    ensures ∀ bi:real; s1, s2, s3 ≤ bi
      ⇒ sensor ≤ 2*bi; */
node voter (s1, s2, s3: real)
  returns (sensor: real);
let
...
tel
```

LUSTRE-C

frama C ..... C code

Summary and future work

Future work

- Develop, develop, develop: support of arrays, matlab scripts, make it work on different NASA use cases
- Example: The NASA TCM (Transport Class Model) use case
- Provide test suite generation for arguing about the validity of the use of the Lustrec compiler
- Intgeration with IKOS (abstract interpretation based static analyzer).

References

- Kind : `http://clc.cs.uiowa.edu/Kind/`
- GeneAuto : `http://geneauto.gforge.enseeiht.fr/`
- [*LCTES'08*] : D. Biernacki, J-L. Colaço, Gr. Hamon, M. Pouzet: *Clock-directed modular code generation for synchronous data-flow languages.* In LCTES 2008.

Detail informations about *CristalCaveGem* contact:

Temesghen Kahsai : temesghen.kahsaiazene@nasa.gov