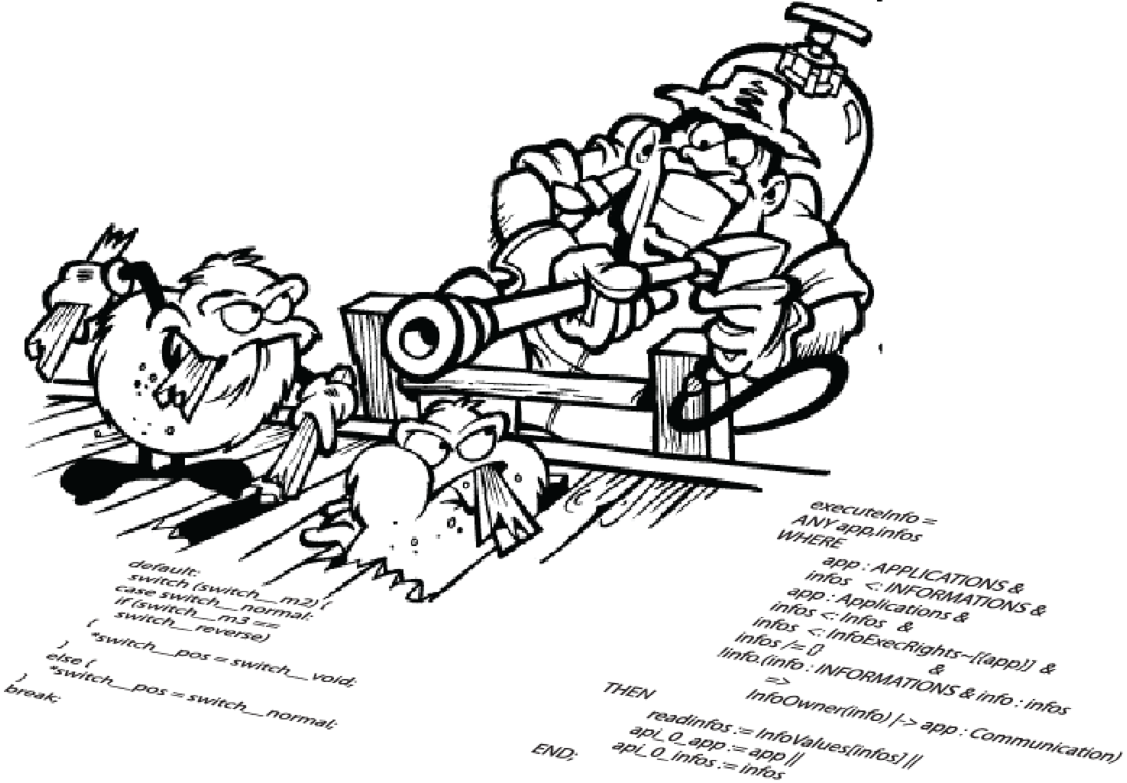


Automatic Refinement and Code Generation - lessons learned -

Thierry Lecomte

thierry.lecomte@clearsy.com

CLEARSY
System Engineering

(*) Inspired from llvm.org/docs/img/Debugging.gif

Plan

- Introduction
- Automatic Refinement
- Code generation
- Perspectives

Introduction

- B model is not end-product
- Hardly readable/understandable even by its creator

> **double dutch**
>
> Totally alien to you, something you don't understand.
>
> *I don't understand this C++ stuff, it's all double dutch to me!*
>
> Urbandictionnary.com

- No processor so far able to natively execute B models

Introduction

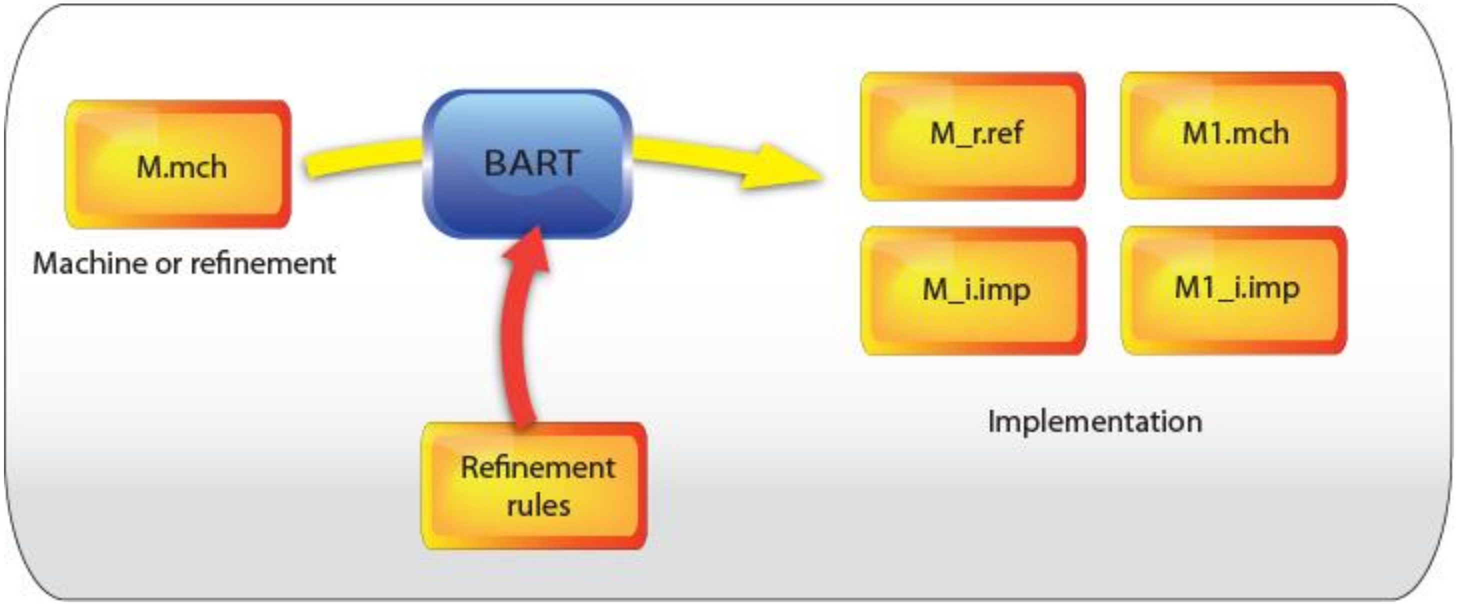
- Hence some transformations are required:
  - Animation
  - (Automatic) Refinement
  - “Code” Generation
- This presentation focuses on last two items

Automatic Refinement

- Refinement is for easing proof
- Why a designer would spend (lost?) time to help a tool doing its job ?
- Expected outcome: time (money) saved when applying the tool
- Errors in the tool are detected when proving the generated models

Setting up methodology and tools (automatic refinement)

- Input: complete set-theoristic model of a software
- Output: refinements and implementations

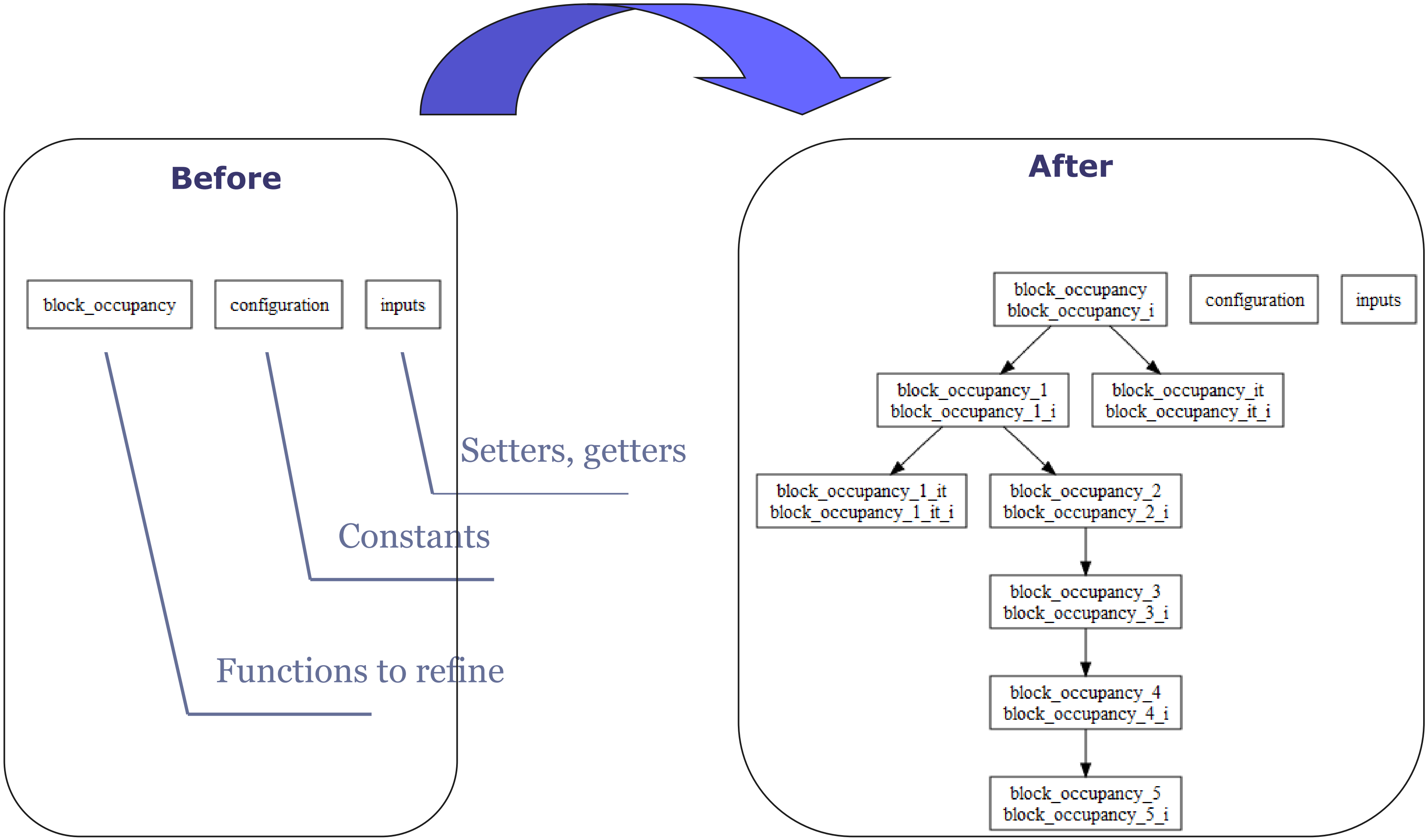


- Refinement engine: applying transformation rules

```
RULE assign_a_bool_subset_b_c_11
REFINES
    @a := bool(@b <: @c-@d)
REFINEMENT
    @a := bool(@b <: @c & @b /\ @d = {})
END;
```

```
RULE assign_a_bool_belongs_b_c_16
REFINES
    @a := bool(@b|->@d : @c*@e)
REFINEMENT
    @a := bool(@b:@c & @d:@e)
END;
```

Setting up methodology and tools (automatic refinement)



Setting up methodology and tools (automatic refinement)

tdla := tdla ∪ ob - mb - otd

Condensed Implementation

```
IMPLEMENT (vg_boucle <-- initier_iteration_t_block) ;
WHILE vg_boucle = TRUE DO
  IMPLEMENT (vg_boucle , l_1 <-- poursuivre_iteration_t_block) ;
  /* ?tdla := tdla \/ ({l_1} /\ ob - mb - otd)? */
  set_tdl_alarm_1([par_in_0_1:=l_1]) =
      [
        BEGIN
          /* ?l_2 := bool(par_in_0_1 : ob - mb - otd)? */
          [par_out_1_1:=l_2] <-- set_tdl_alarm_1_1([par_in_1_1:=par_in_0_1]) =
              [
                BEGIN
                  l_5 := ob_i ( par_in_1_1) ;
                  l_7 := mb_i ( par_in_1_1) ;
                  l_6 := bool(l_7 = FALSE) ;
                  l_3 := bool(l_5 = TRUE &
                  l_6 = TRUE) ;
                  l_8 <-- lire_otd(par_in_1_1) ;
                  l_4 := bool(l_8 = FALSE) ;
                  par_out_1_1 := bool(l_3 = TRUE &
                  l_4 = TRUE)
                END
              ] ;
          IF l_2 = TRUE THEN
            /* ?tdla := tdla \/ {par_in_0_1}? */
            set_tdl_alarm_1_2([par_in_1_1:=par_in_0_1]) =
                [
                  BEGIN
                    tdla_i ( par_in_1_1) := TRUE
                  END
                ]
          END
        END
      ]
INVARIANT
  vg_boucle = bool(t_block_a_traiter /= {}) &
  t_block_a_traiter \/ t_block_traites = t_block &
  t_block_a_traiter /\ t_block_traites = {} &
  tdla = tdla$0 \/ t_block_traites /\ ob - mb - otd
```

Refinement tree

```
Refinement rules
◢ Bloc substitution
  ◢ assign_a_union_b_c.assign_a_union_b_c_1
    ◢ set_tdl_alarm_1 : Guarded substitution
      ◢ block.R4
        ◢ set_tdl_alarm_1_1 : Guarded substitution
          ◢ assign_a_bool_belongs_b_c.assign_a_bool_belongs_b_c_21
            ◢ assign_a_bool_prop_op_b_c.assign_a_bool_and_b_c_1
              ◢ Semicolon
                ◢ Semicolon
                  ◢ assign_a_bool_belongs_b_c.assign_a_bool_belongs_b_c_21
                    ◢ assign_a_bool_prop_op_b_c.assign_a_bool_and_b_c_1
                      ◢ Semicolon
                        ◢ Semicolon
                            assign_a_bool_belongs_b_c.assign_a_bool_belongs_b_c_39
                          ◢ assign_a_bool_not_b.R1
                            ◢ Semicolon
                                assign_a_bool_belongs_b_c.assign_a_bool_belongs_b_c_39
                                assign_a_bool_eq_b_c.assign_a_bool_eq_b_c_1
                          IMPLEMENT
                  ◢ assign_a_bool_not_b.R1
                    ◢ Semicolon
                        assign_a_bool_belongs_b_c.assign_a_bool_belongs_b_c_40
                        assign_a_bool_eq_b_c.assign_a_bool_eq_b_c_1
                IMPLEMENT
        ◢ set_tdl_alarm_1_2 : Guarded substitution
            assign_a_union_b_c.assign_a_union_b_c_2
```

Setting up methodology and tools (automatic refinement)

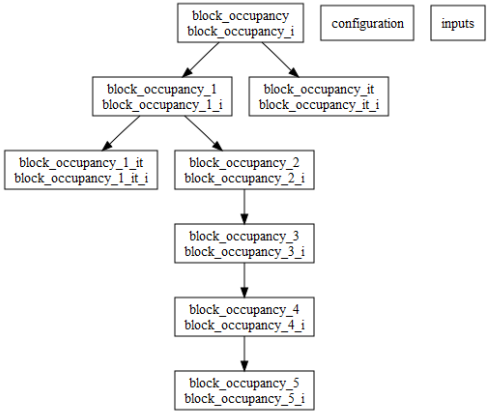
- Outcomes: development time divided by 2
- Safety Critical Software usually require twice more workload
- SCS developed with non SCS budget
- 700 refinement rules written down
- Deployed worldwide for several metros

- Biggest implementation: Val de Roissy Shuttle
  - Alarm Control Unit: 265 kloc B model (40 kloc handwritten), 186 kloc Ada code
  - Section Automatic pilots: 67 kloc B model, 50 kloc Ada code

Automatic Refinement

- How is it practical ? (-> LIVE DEMO)

- How is it efficient ?
  - Generated models are more decomposed
  - Many small steps leading to easier proof
  - For some constructions (abstract iterator), interactive demonstration could be provided automatically



Pattern matching in detail

Hypothesis | Node substitution | Node rule | Matching rules

Substitution to refine

```
l_16 := bool(par_in_1_1 : otd)
```

Matching rule

```
RULE assign_a_bool_belongs_b_c_40
REFINES
  @a := bool(@b : @c)
WHEN
  DECL_OPERATION(@d <-- @e(@f) |
    PRE
      @g
    THEN
      @d := bool(@f : @c)
    END)
IMPLEMENTATION
  @a <-- @e(@b)
END
```

Matching getter

```
res <-- lire_otd(p_block) =
PRE
      p_block : t_block
THEN
      res := bool(p_block : otd)
END;
```

Refined substitution

```
l_16 <-- lire_otd(par_in_1_1)
```

Feedback

- Initial set of refinement rules is not sufficient
  - Need to be extended to address your modelling and expectations
- Initial set of rules is not bug free
  - Detected by typechecking (syntax errors) and by proof

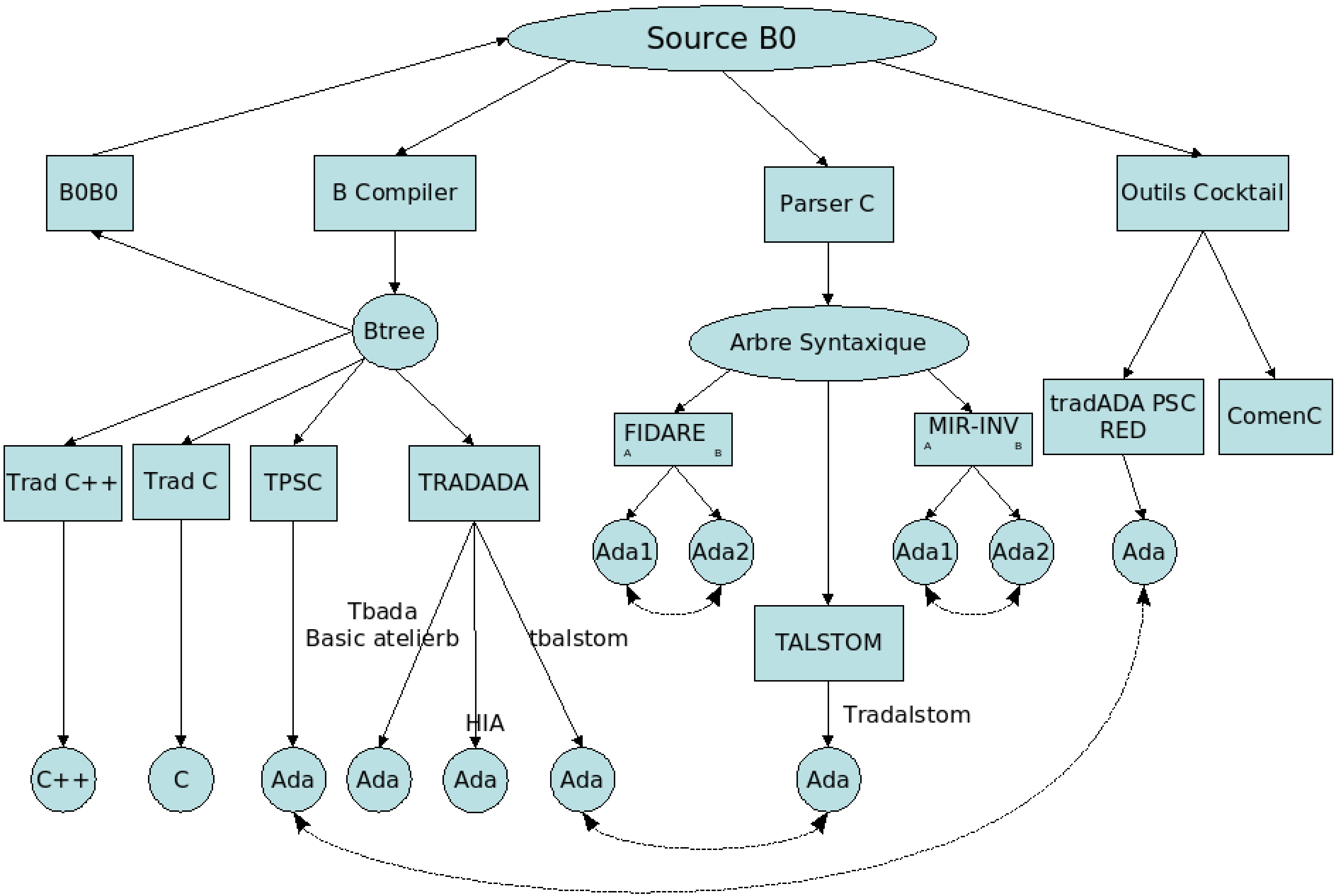
Code generation

- Several code generators in use: C, C++, Ada, HIA
  - Incoming Ladder and VHDL

- Using different technologies (redundancy)
  - Encoding (FIDARE),
  - diversity (inverse mirror),
  - specific hardware (coded secure processor)

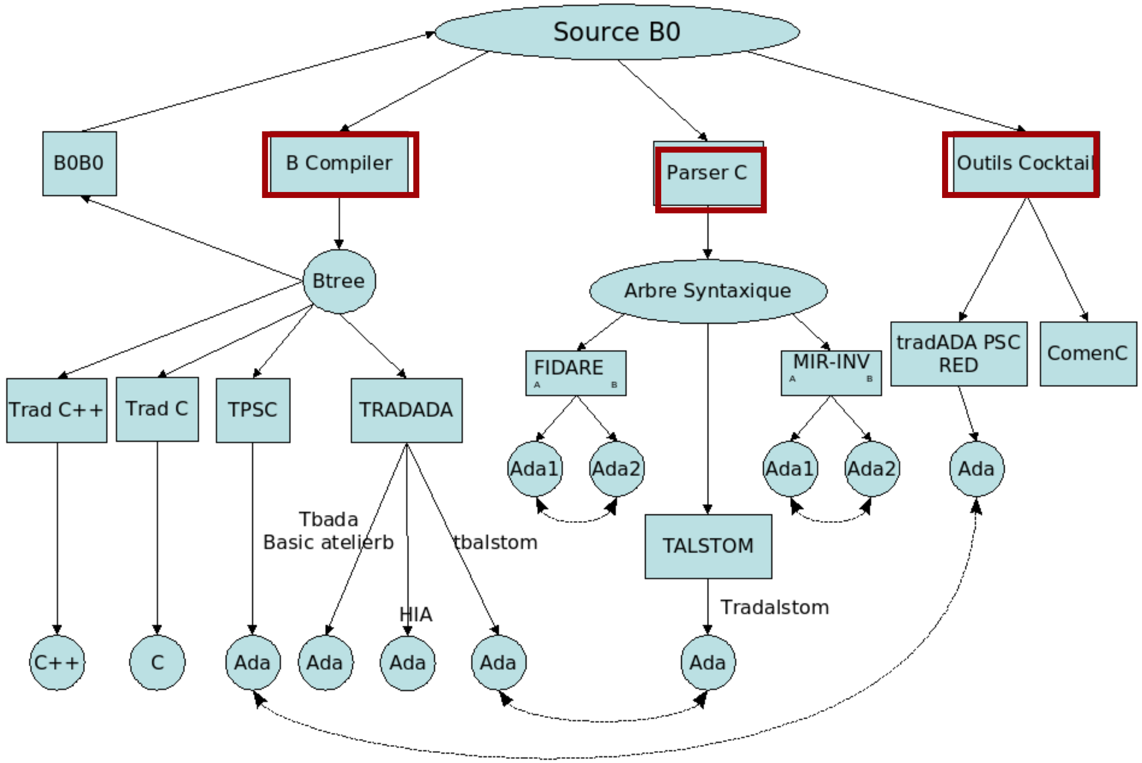
- Almost each industrial project has its own translator

- The current situation is .....


Source B0
B0B0
B Compiler
Parser C
Outils Cocktail
Btree
Arbre Syntaxique
Trad C++
Trad C
TPSC
TRADADA
FIDARE
A
B
MIR-INV
A
B
tradADA PSC RED
ComenC
Ada1
Ada2
Ada1
Ada2
Ada
Tbada
Basic atelierb
tbalstom
TALSTOM
Tradalstom
HIA
C++
C
Ada
Ada
Ada
Ada
Ada

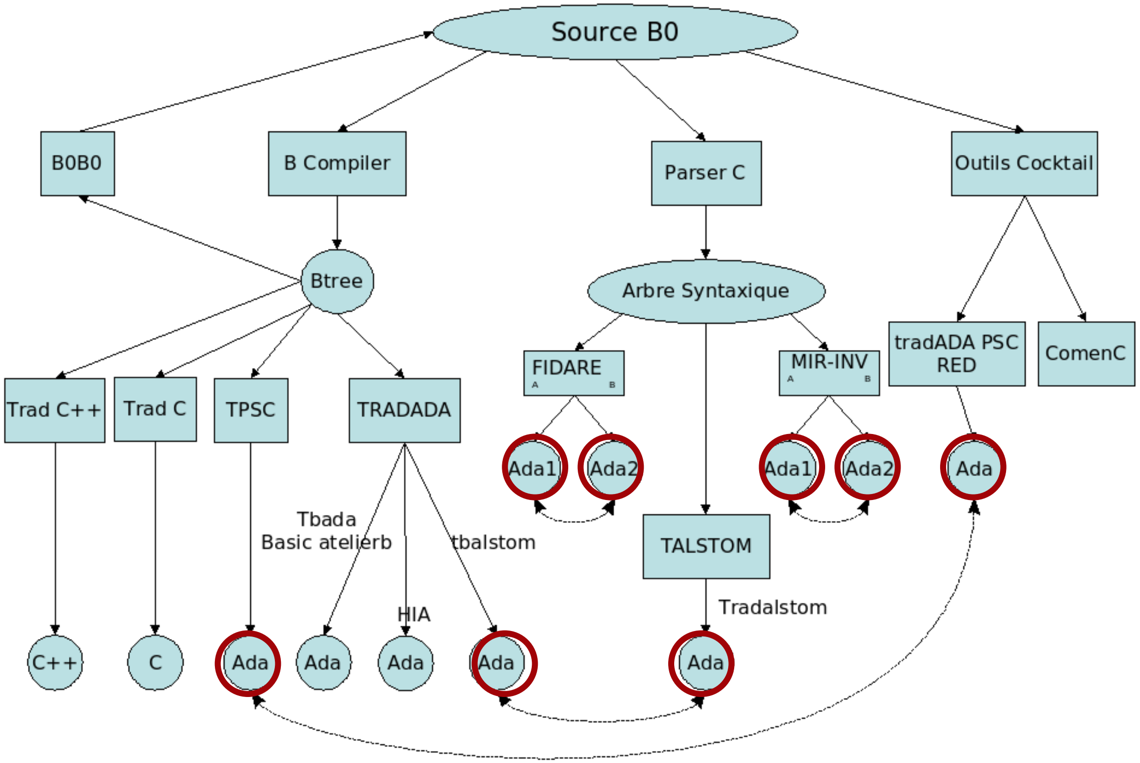
Code generation

- Based on different tools to avoid common mode failure
  - Type-checker, B-parser, B-compiler



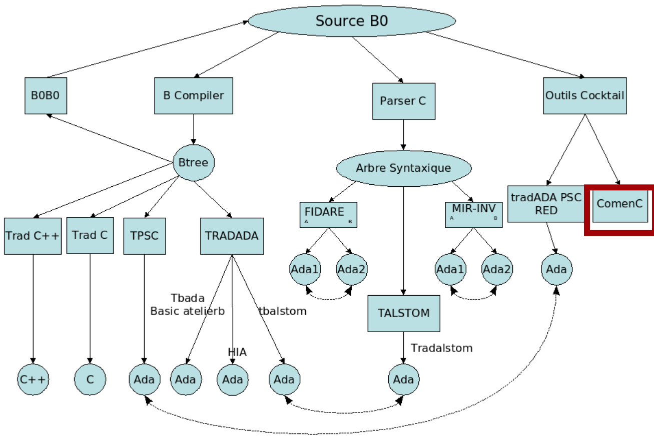
Code generation

- Translators to be used in pair



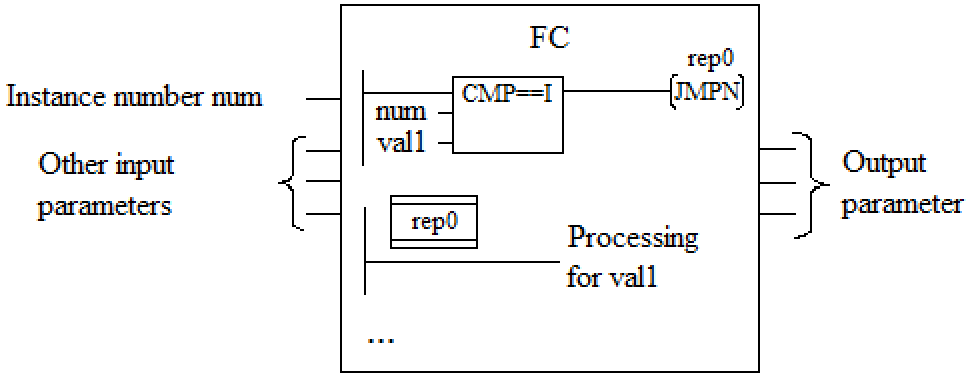
C Code generation

- Safety critical standards recommend:
  - (1) A limited use of pointers
  - (2) No recursion
  - (3) No dynamic memory allocation
- With instantiated machines, point (1) was not reachable
- Development of a translator based on cocktail compiler compiler: ComenC
- C code more readable
- But discontinued support



Ladder Code Generation

- Transformation of a B model into a ladder code in order to feed a PLC

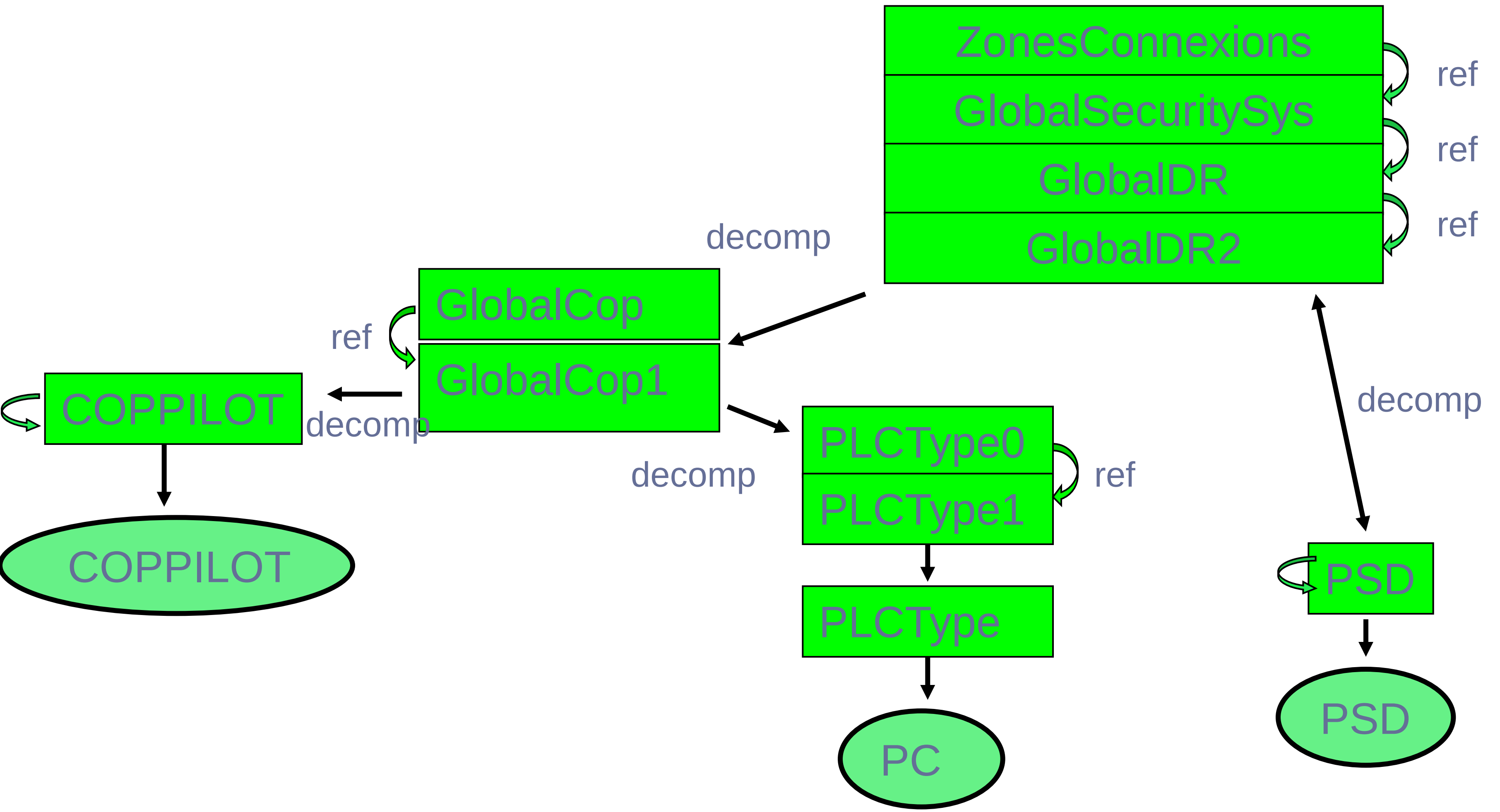


- For S7 Simatic (Siemens)
- Generation of png files !

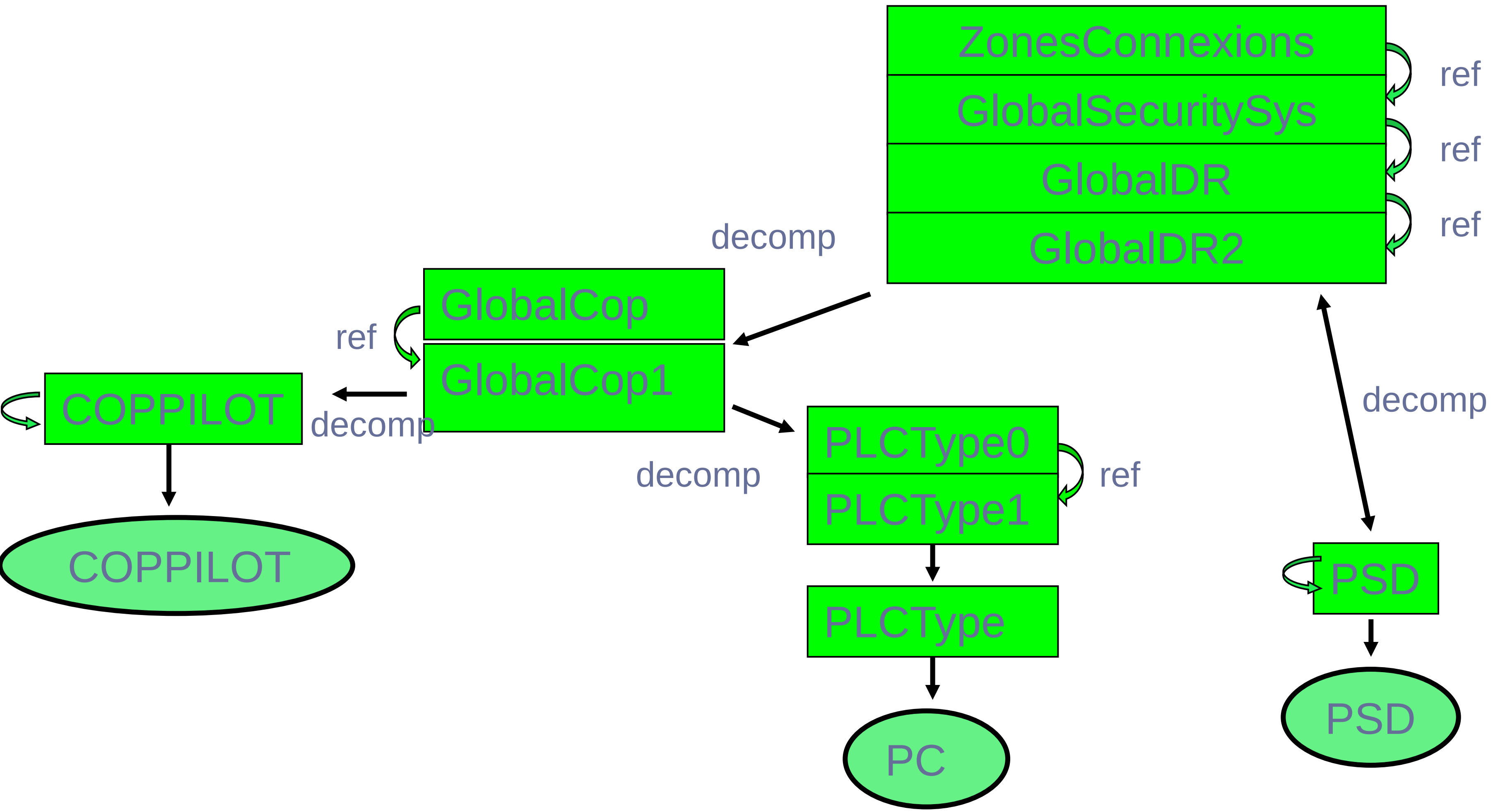
Modelling phase 1

- define the properties expressing system safety
- demonstrate that any train + PSD system veryfing some properties is safe
- open train doors iff train is at the standstill and doors in front of PSD
    - open PSD iff train at the standstill is present or in case of evacuation
    - a train should not move if at least one PSD is not closed

Modelling phase 2



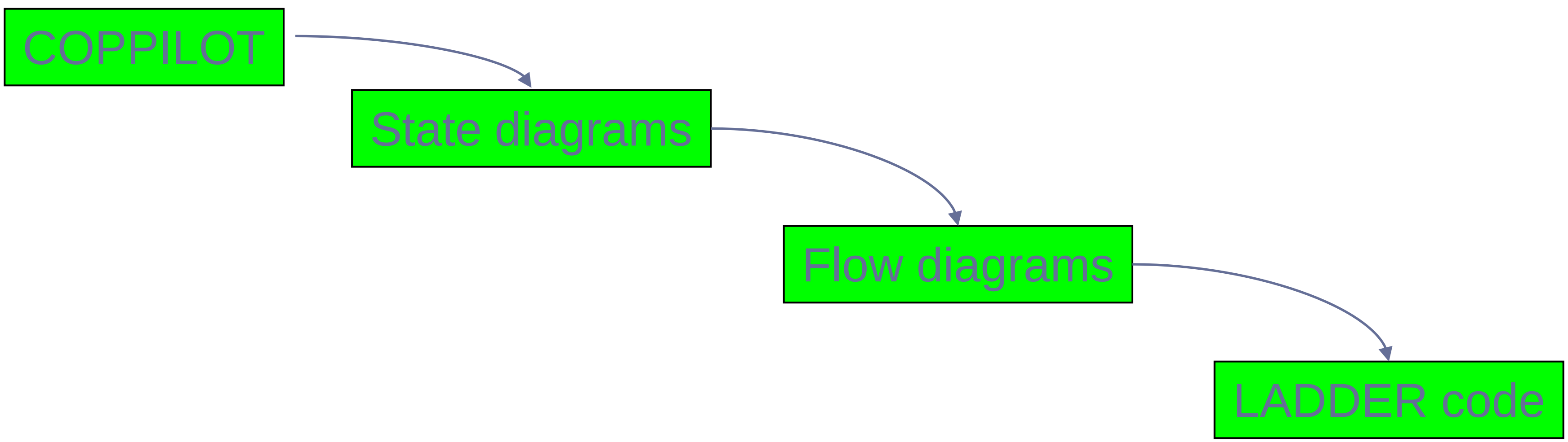
Modelling phase 2



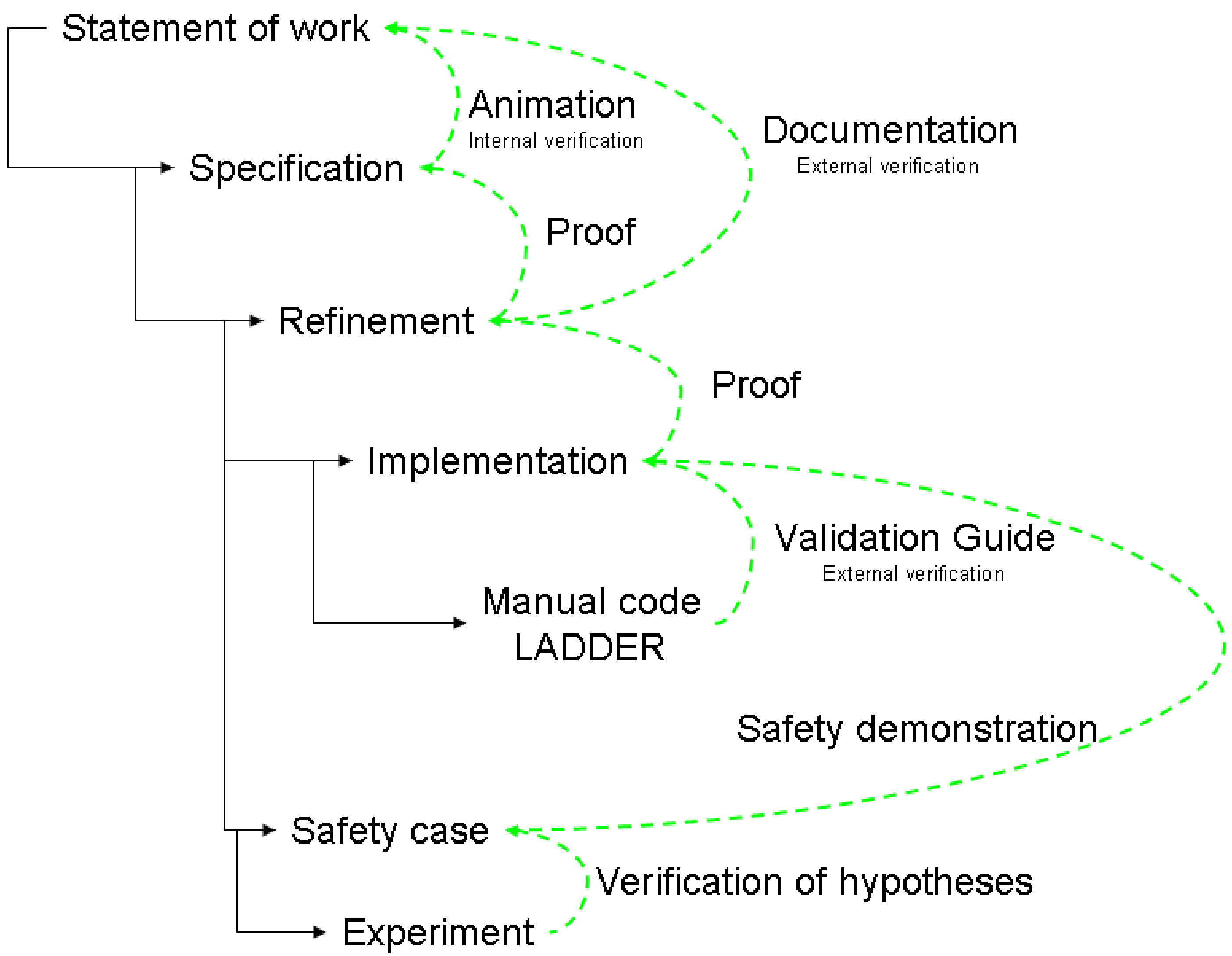
Modelling phase 3

```
1.  ----[  ]----------|--[  ]--|-------(  )--
         X            |    Y   |         S
                      |        |
                      |--[  ]--|
                           Z
```

The above realises the function: S = X AND (Y OR Z)



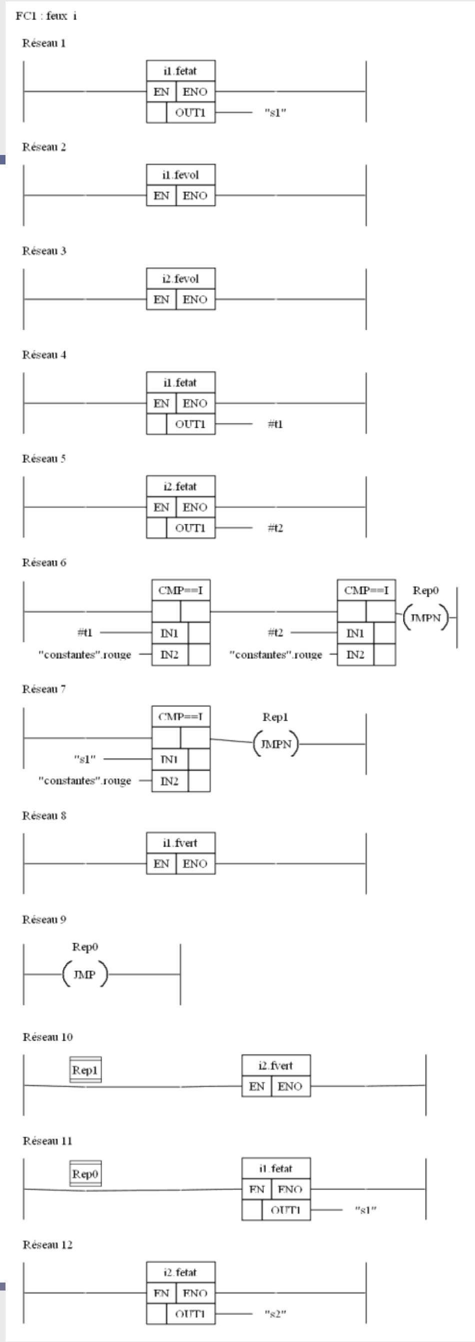
Verifications



Traffic light management

```
s1,s2 <-- evol =
VAR t1,t2 IN
    s1 <-- i1.fetat;
    i1.fevol;
    i2.fevol;
    t1 <-- i1.fetat;
    t2 <-- i2.fetat;
    IF t1=rouge & t2=rouge THEN
        IF s1=rouge THEN
            i1.fvert
        ELSE
            i2.fvert
        END
    END;

    s1<--i1.fetat;
    s2<--i2.fetat
END
```



Feedback

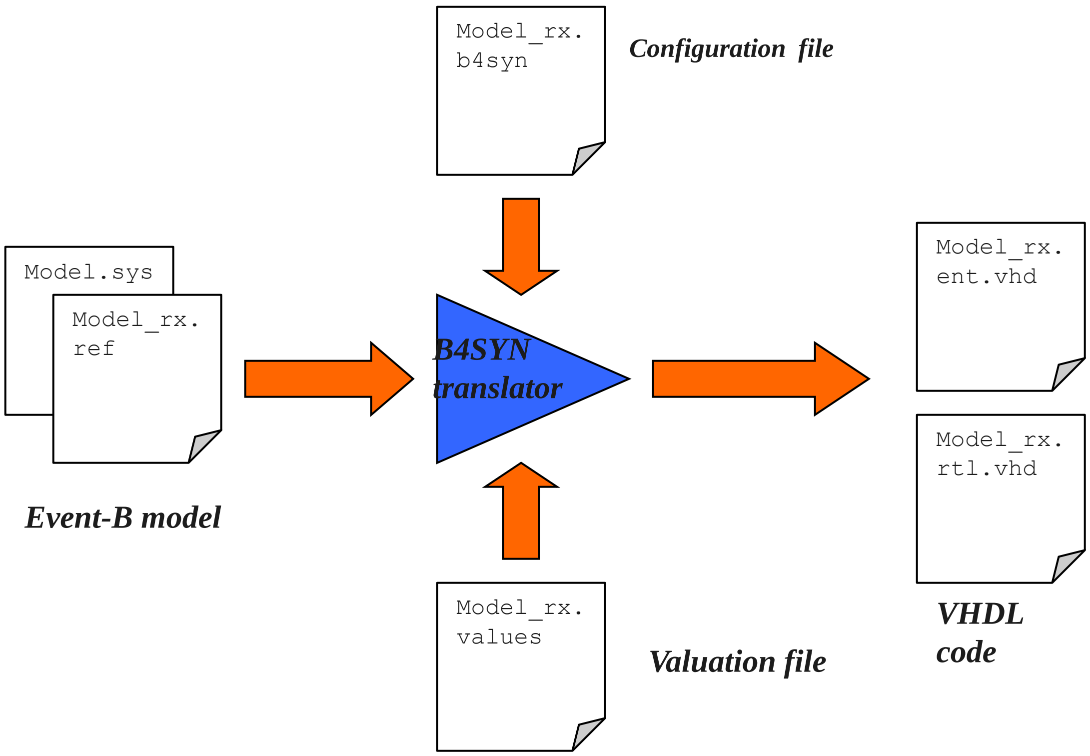
- *Applied several times for safety critical systems*
- *Typing Ladder programs using SIMATIC S7 PLC is risky*
- *Envisaging to directly generate binary code*

VHDL Code Generation (B4SYN)

- Not a 1-to-1 translation schema

- *What is translated*
  - *Variables*
  - *Constants*
  - *Events*
- *What is needed*
  - *Invariants*
  - *Properties*
  - *Valuation for the constants*
  - *List of synchronous events and outputs*
  - *List of asynchronous events*
  - *List of combinatorial events and outputs*

B4Syn Translation schema



Extra information

- *Synchronous events*: modelling computations performed on a clock tick.
  - The inputs are acquired
  - Outputs are positioned
  - Registers are updated

- *Asynchronous events*: modelling interrupted events
  - Registers should be initialized
  - cold reset, warm reset

- *Combinatorial events*: events triggered before the component stabilizes.
  - is necessary to check that the disjunction of their guards is true.

- Supported grammar:

  [*ASYNCHRONOUS* | (*SYNCHRONOUS* ; *ASYNCHRONOUS*) | (*SYNCHRONOUS* ; *COMBINATORIAL*)]*

Extra information

- *Circuit definition*
  - *Synchronous outputs*
  - *Combinatorial outputs (handled in a separate process)*
  - *Clocks*
  - *Sequencer variable*
  - *Inputs*
- *Valuation information (sets, functions, relations, integers, elements)*
  - *Values used for driving the translation process*
    - *S = a..b indicates that the set is a range of values from a to b*
    - *S = (0..b)*BIT indicates that the set is a range of bits, and that bit to bit operations are possible*
  - *Predefined functions:*
    - $NthBit = \lambda(x,n).(x \in (0..i)*BIT \wedge n \in 0..i \mid x(n))$

VHDL types supported

- *STD_LOGIC*
- *STD_LOGIC_VECTOR(x downto y)*
- *INTEGER*
- *Arrays of the previous types*

Structure of the rtl file

- *Process sample inputs*
- *Process registers reset*
- *Process output management*
- *Combinatorial events*

Feedback

- *Translator used with success on a microciruit*
- Adequate generated VHDL models:
  - Size (5k gates)
  - Workload (even if different profiles)
  - Able to be tested with product testbenches
- *Translator probably lacking of generality*

Generating Ada code from Event B model

- *Application of aggregation rules to transform a set of events into an algorithm*

```
SELECT P ∧ Q THEN R END
[]
SELECT P ∧ not Q THEN S END
~>
SELECT P THEN
   IF Q THEN R ELSE S END
END
```

Condition:

P ∧ Q => [R] not P

P ∧ Q => [S] not P

Generating Ada code from Event B model

```
SELECT P THEN R END
[]
SELECT Q THEN S END
~>
SELECT P THEN R;S END
```

Condition:

P => [R] Q

Generating Ada code from Event B model

- *Obtained algorithm is not checkable with B*
- *Applied on part of the Ariane 5 flight software*
- *To obtain finally 80 lines of Ada, comparable to the handwritten ones*
- *Around 20 000 events would be required to replicate the branching structure of an Automatic Train Pilot*

Semantics of B models

```
IF xx + yy > 255 THEN
    xx := xx mod 2
END
```

→ Refined model

→ C

→ Ada

→ Ladder

→ VHDL

Would my modelling be the same for a different target formalism ?

Conclusion

- Path to cyclic software well explored
- Different approaches for event based models, even not 1 to 1 translation
- Still lot to do

Thank you for your attention