Proven Software with

Executing on

Low Cost High Integrity Platform

Thierry Lecomte
October 20, 2016
Sherbrooke

CLEARSY
SYSTEM ENGINEERING

Agenda: B formal method,high integrity, safety critical

Introduction

- Safety critical systems & standards

Proven software with B

- B (101), Proven with B, Software with B
- Improved “improved development cycle”

Low Cost High Integrity Platform

- Double processor, double toolchain, double software instances
- LCHIP

Introduction

Safety Critical Systems

Systems where life is at risk

Should either work properly or avoid to kill people in case of failure

- Default-free is not from this earth
- Safety case as demonstration

Safety Critical Systems

Standards

Highest level

• Railways	EN5012{6, 8, 9}	SIL	0, 1, 2, 3, 4
• Aeronautics	DO-178C	DAL	E, D, C, B, A
• Automotive	ISO 26262	ASIL	0, 1, 2, 3, 4
• Industry	IEC 61508	SIL	0, 1, 2, 3, 4

SIL3: 10^{-7} failure / hour

SIL4: 10^{-9} failure / hour

µprocessor: 10^{-5} failure / hour

SIL3, SIL4: 2 µprocessors (or more) required

(2oo2, 2oo3, voter, processor+coprocessor, etc.)

Safety Critical Systems

For highest SIL, formal methods are highly recommended … as well as other means

We are far from

"*It compiles hence it works !*"

IEC 61508: Software design and dev. (table A.2)

Technique/Measure	Ref	SIL1	SIL2	SIL3	SIL4
1 Fault detection and diagnosis	C.3.1	---	R	HR	HR
2 Error detecting and correcting codes	C.3.2	R	R	R	HR
3a Failure assertion programming	C.3.3	R	R	R	HR
3b Safety bag techniques	C.3.4	---	R	R	R
3c Diverse programming	C.3.5	R	R	R	HR
3d Recovery block	C.3.6	R	R	R	R
3e Backward recovery	C.3.7	R	R	R	R
3f Forward recovery	C.3.8	R	R	R	R
3g Re-try fault recovery mechanisms	C.3.9	R	R	R	HR
3h Memorising executed cases	C.3.10	---	R	R	HR
4 Graceful degradation	C.3.11	R	R	HR	HR
5 Artificial intelligence - fault correction	C.3.12	---	NR	NR	NR
6 Dynamic reconfiguration	C.3.13	---	NR	NR	NR
7a Structured methods including for example, JSD, MASCOT, SADT and Yourdon	C.2.1	HR	HR	HR	HR
7b Semi-formal methods	Table B.7	R	R	HR	HR
7c Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	C.2.4	---	R	R	HR
8 Computer-aided specification tools	B.2.4	R	R	HR	HR

a) Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

b) The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in part 2 of this standard.

Safety Critical Systems

Ability to command outputs

- Bijection between processor memory and actuator physical state
- Checked often
- Restrictive mode when check fails (reboot, shutdown, etc.)

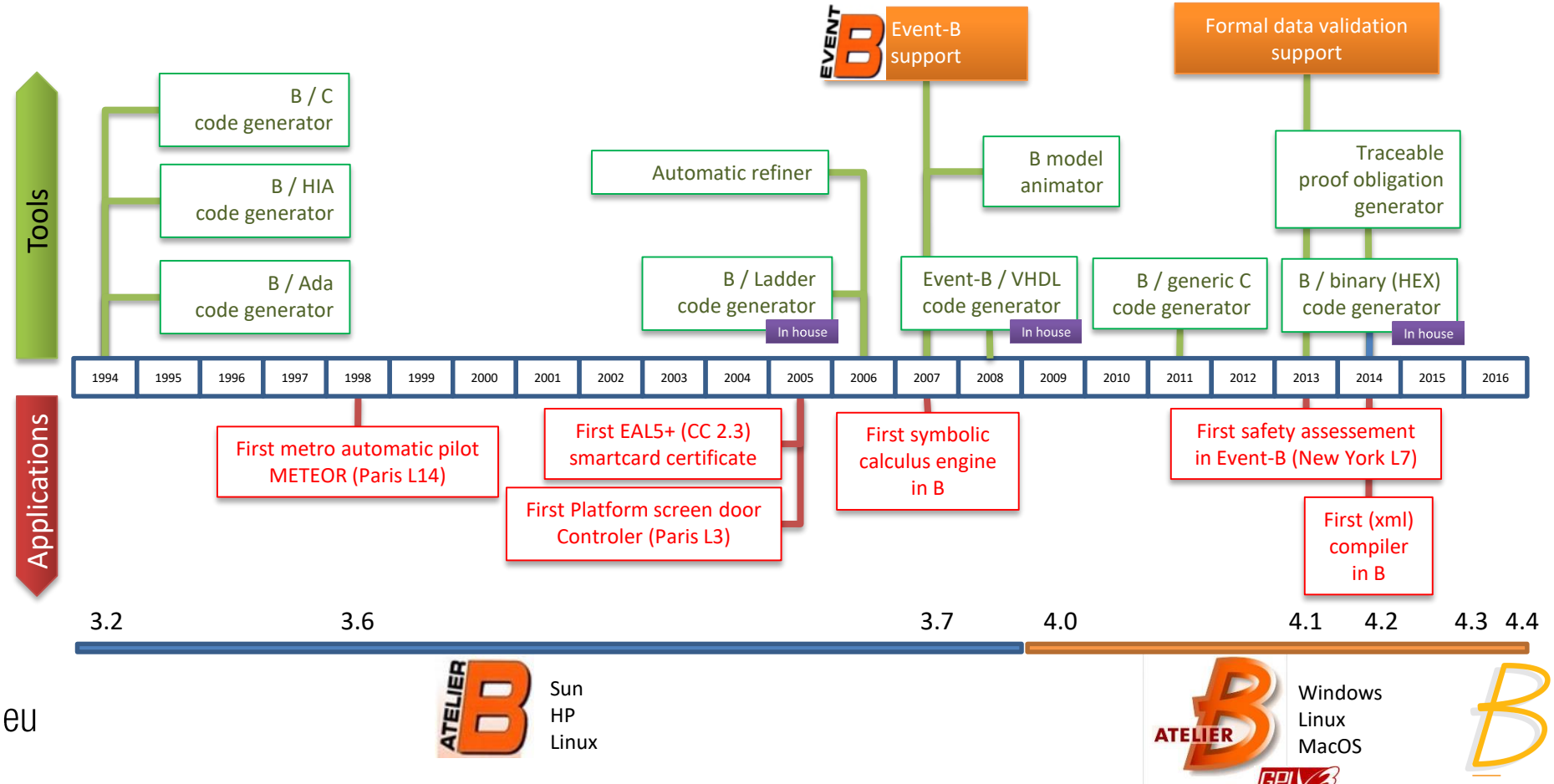
Restrictive/degraded mode

- No universal RM, depends on the system
- Trains: usually reduce kinetic energy (emergency breaking)
- Planes: give commands back to human pilot

Proven Software with B & Contribution to Safety Critical Systems

Atelier B initially developed by Fernando Mejia

- Development funded for Driverless metro in Paris (L14)
  - First customer: University of Sherbrooke
    - METEOR released
      - Creation of ClearSy (full ownership)
        - Atelier B Community + Maintenance



www.atelierb.eu

Disclaimer

- Demos are made with not yet public Atelier B versions (4.4 & 4.5)
- Current version 4.3 can be downloaded at www.atelierb.eu

B (101)

- formal method (semantics)
  - text-based models of unthreaded programs
  - same language to specify **(MACHINE)** and design **(IMPLEMENTATION)**
  - set theory and first order logic
  - Specification for the “what”
  - Design for the “how” and the architecture

```
MACHINE
    M0
VARIABLES
    xx, ii
INVARIANT
    xx: BOOL &
    ii : INTEGER &
    (ii > 0 => xx = TRUE)
INITIALISATION
    ii :=0 ||
    xx := TRUE
OPERATIONS
    update =
    BEGIN
        xx := TRUE ||
        ii :: 1..3
    END
END
```

B (101)

Variables

- Can be abstract (not implemented), just for reasoning
- Can be concrete: need to be implemented in one and only one implementation

Invariant

- Types for variables (set, function, scalar, boolean, integer, table, etc.)
- Constraints for variables
- Invariant is true all the time
  - ensured by initialisation
  - If true before executing an operation, still true after

Initialisation

- First value for variables
- Can be non-deterministic
- All variables initialized at the same time

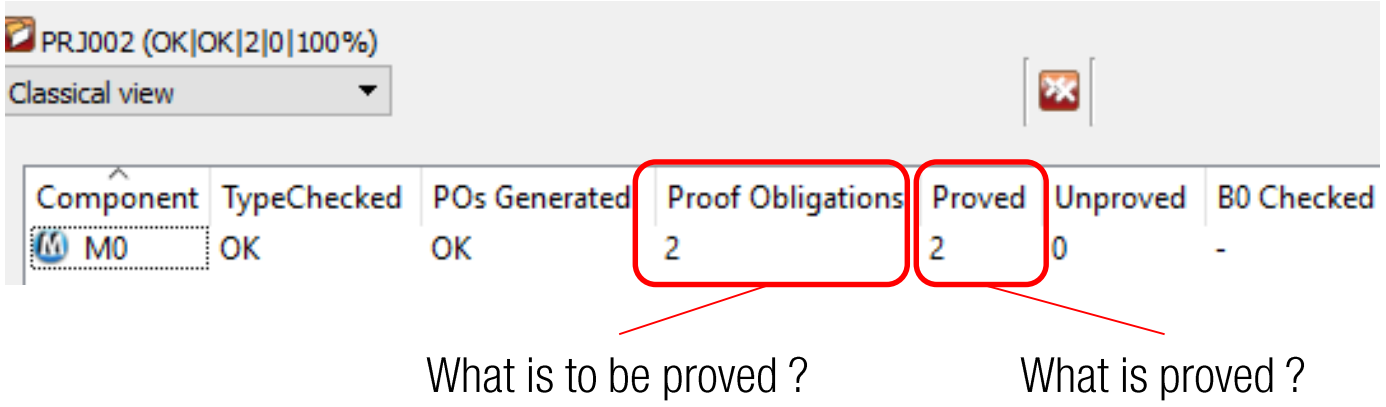
Operation

- Modify modelling variables
- A precondition may specify under which conditions the operation is callable
- Specified as a transfer function (no algorithm)

```
MACHINE
    M0
VARIABLES
    xx, ii
INVARIANT
    xx: BOOL &
    ii : INTEGER &
    (ii > 0 => xx = TRUE)
INITIALISATION
    ii :=0 ||
    xx := TRUE
OPERATIONS
    update =
    BEGIN
        xx := TRUE ||
        ii :: 1..3
    END
END
```

B (101)

- with proof (models are validated by proof)
  - coherency / compliancy expressed as proof obligations
  - proof obligations are not an option
  - automatic generation
  - Pro: proof equivalent to exhaustive testing
  - Con: proof is semi-automatic (require expert transactions to complete)
  - Proof extent can be checked at a glance

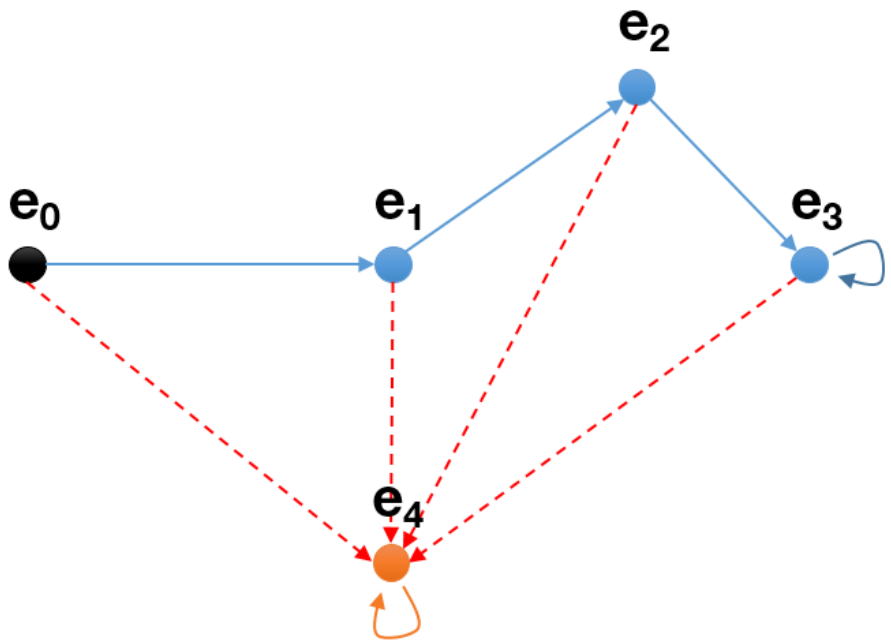



```
MACHINE
    M0
VARIABLES
    xx, ii
INVARIANT
    xx: BOOL &
    ii : INTEGER &
    (ii > 0 => xx = TRUE)
INITIALISATION
    ii :=0 ||
    xx := TRUE
OPERATIONS
    update =
    BEGIN
        xx := TRUE ||
        ii :: 1..3
    END
END
```

```
"Invariant is preserved" &
"Check invariant ((ii) : (INTEGER))"
=>
0: INTEGER
```

```
ii$1: 1..3 &
"Invariant is preserved" &
"Check invariant ((ii) : (INTEGER))"
=>
ii$1: INTEGER
```

Example: implementing a state machine




```
MACHINE
    CTX
SETS
    STATES = {e0, e1, e2, e3, e4}
CONSTANTS
    next
PROPERTIES
    next = {
        e0 |-> e1,
        e1 |-> e2,
        e2 |-> e3,

        e3 |-> e3,

        e0 |-> e4,
        e1 |-> e4,
        e2 |-> e4,
        e3 |-> e4,
        e4 |-> e4
        }
END
```

Example: implementing a state machine

```
MACHINE
    P0
SEES CTX

VARIABLES
    current_state
INVARIANT
1/1     current_state : STATES
INITIALISATION
    current_state := e0
OPERATIONS
    evolve =
    BEGIN
1/1         current_state :: next[{current_state}]
    END

END
```

Variables

- Current state of the state machine

Variables

- Next current state should belong to next definition

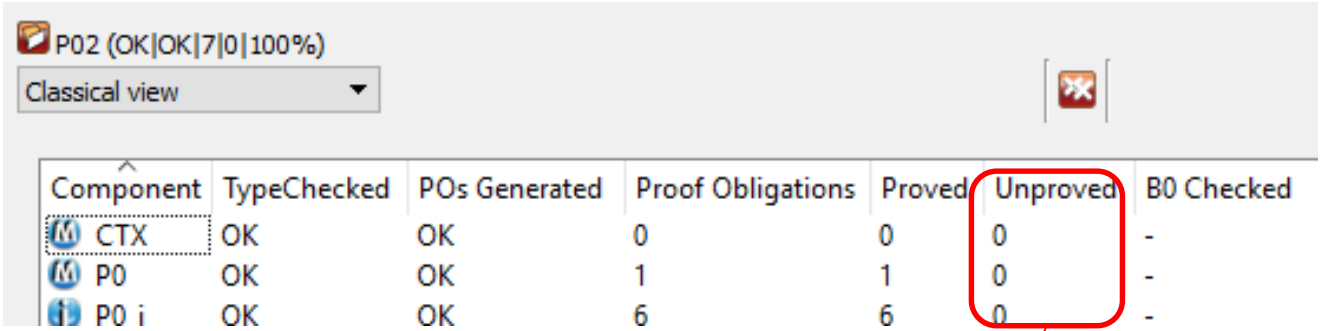
Example: implementing a state machine

```
IMPLEMENTATION P0_i
REFINES P0

SEES CTX

CONCRETE_VARIABLES
6/6     current_state
INITIALISATION
    current_state := e0

OPERATIONS
    evolve =
    BEGIN
6/6     CASE current_state OF
6/6         EITHER e0 THEN current_state := e1
1/1         OR e1 THEN current_state := e2
1/1         OR e2 THEN current_state := e3
1/1         OR e3 THEN current_state := e3
            OR e4 THEN skip
            END
        END
    END
END
```



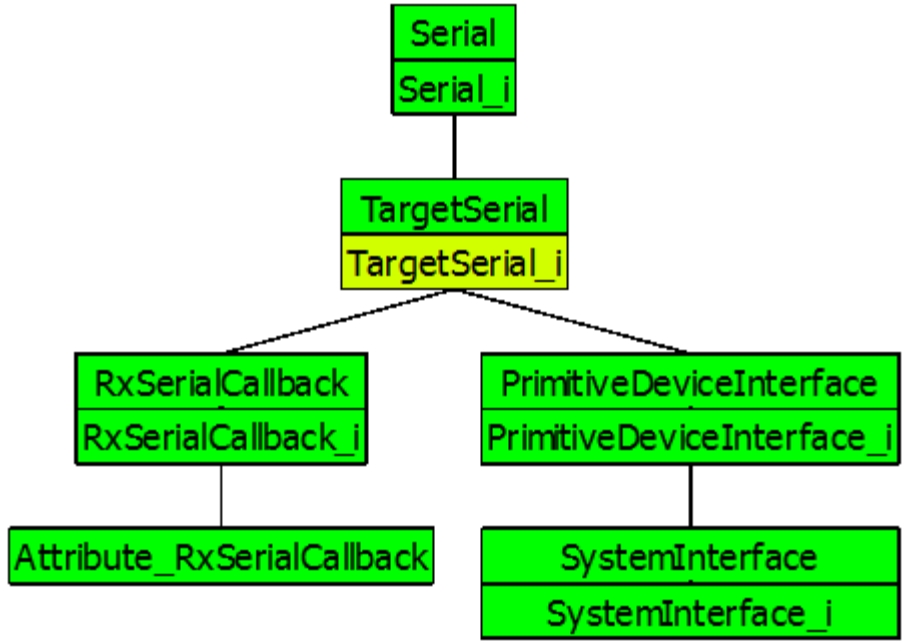
Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
CTX	OK	OK	0	0	0	-
P0	OK	OK	1	1	0	-
P0_i	OK	OK	6	6	0	-

Operations

- An example of a correct implementation
- Proof is semi-automatic though

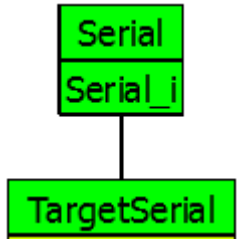
B (101)

- Architecture
  - Based on modules **(MACHINE + IMPLEMENTATION)**
  - An operation can't call another operation of the same module
  - Hence operations have to call other operations from imported components
  - Some components have no implementation in B (developed manually)



Tree-like structure of a B project
(Import links)

  - Proving an **IMPLEMENTATION** requires:
    - Its specification **MACHINE**
    - The specification of the **MACHINE**s it **IMPORTS**
    - The specification of the **MACHINE**s it **SEES**



To prove Serial_i, TargetSerial_i is not required as it complies with TargetSerial

Software with B

In a B project, only implementations are translated

- Implementation have to be implementable
- No more non-determinism, sets, functions, etc.
- Only implementable types (BOOL, INT, tables of BOOL and INT)
- Only imperative constructs: valuation, IF THEN ELSE, CASE, WHILE, sequence, operation call.

Translated code is very close to original B implementation

Software with B

Specification can be translated to obtain code skeleton

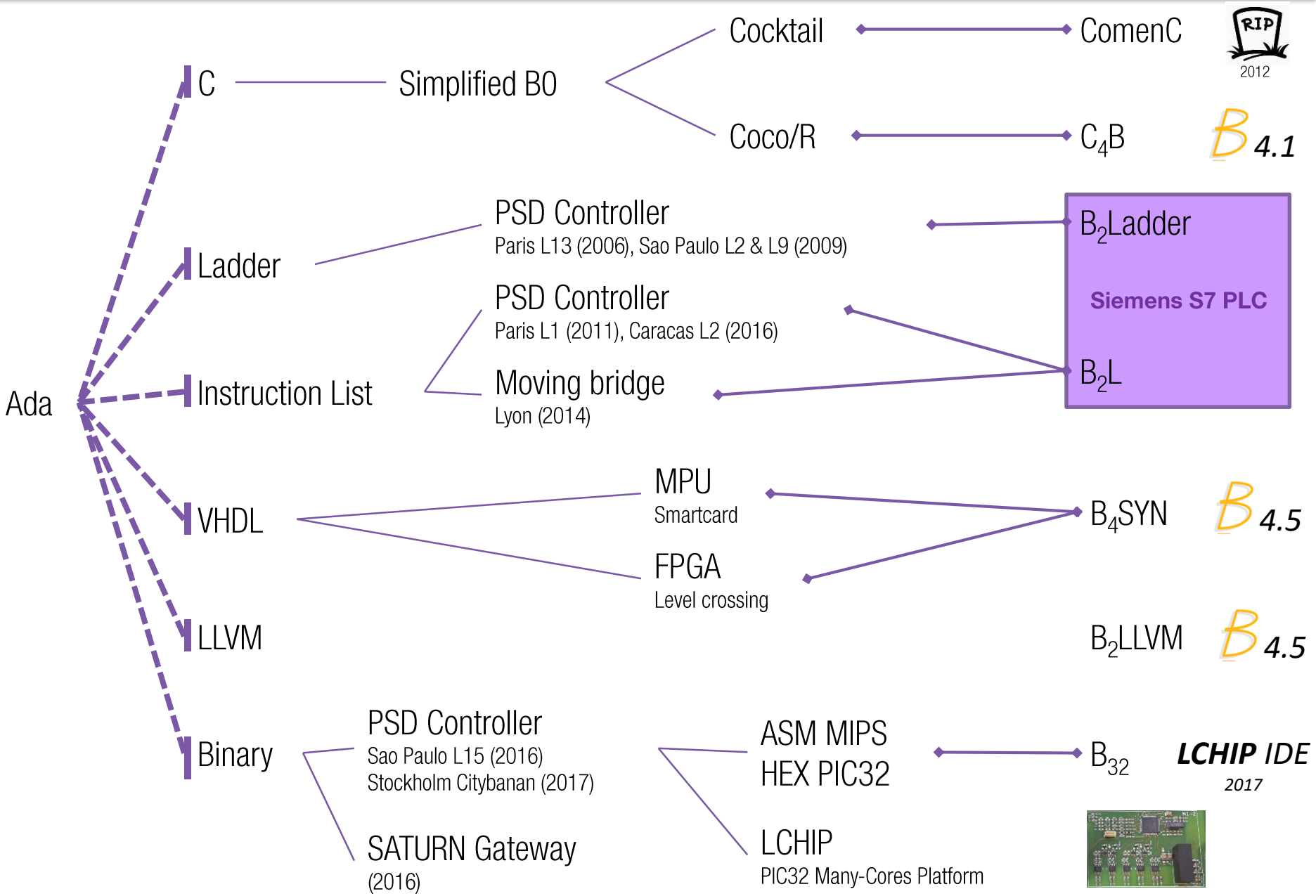
when implementation is

- Manual
- Third-party

B software can:

- be merged / integrated into “traditional” development
- reuse OSS libraries

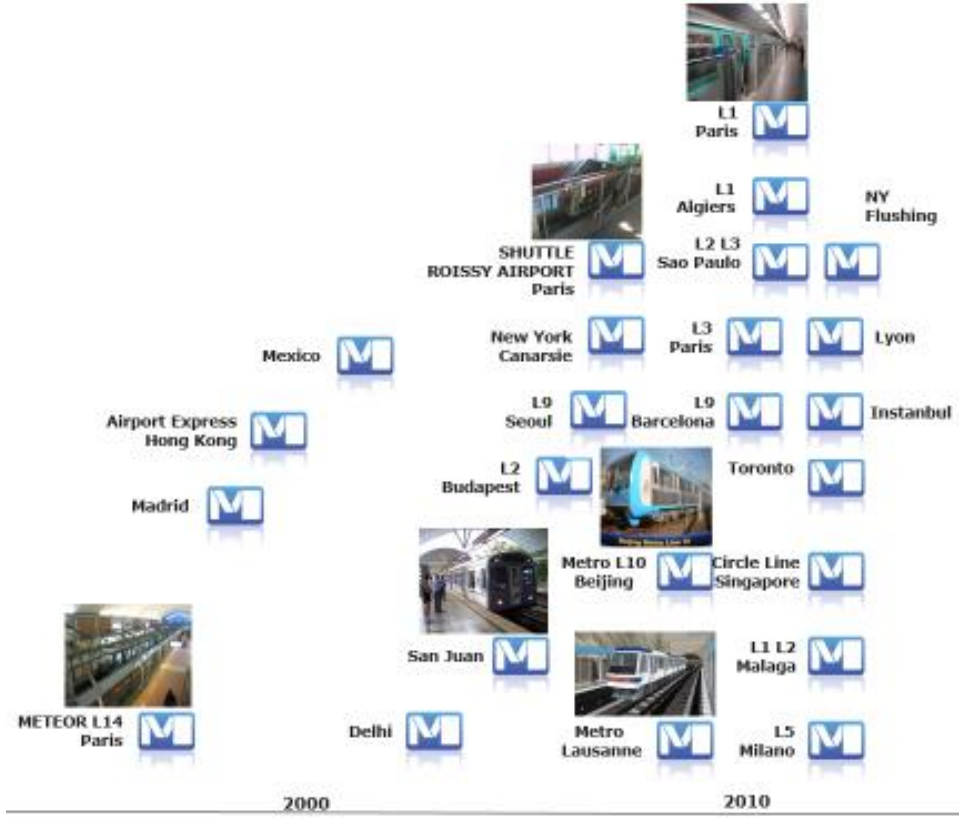
Generating Code for Safety Critical Applications



Ada

≡ Long sequence of implementations

- Paris L14 (1998) : 80 kloc
- Paris Roissy Airport Shuttle (2006): 186 kloc onboard, 50 kloc trackside
- Lille (2015): 300 kloc xml compiler



≡ Using different technologies (redundancy)

- encoding (FIDARE): 2 instances
  - inputs, outputs, variables, instructions are encoded
  - the coding key is different for each instance

- diversity (inverse mirror): 2 instances
  - one instance uses big endian model,
  - the other one executes little endian model

- specific hardware (coded secure processor): 1 instance
  - Variables have two fields: value and code. Instructions (OPELS) modified both accordingly. Mismatching value and code indicates a memory corruption.
  - Compensation tables contains tags for all possible path (masks). An unexpected value indicates a program counter corruption


```
z := x + y;
y := 0
```

```
Z := OPEL_PLUS(X,Y);
OPEL_NULL(Y)
```

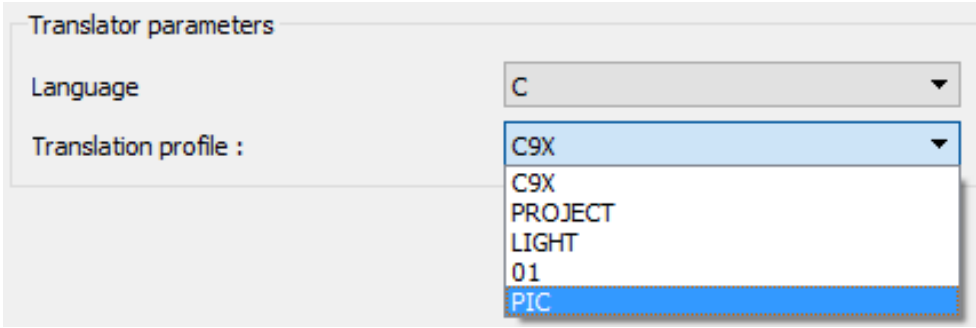
C

≡ Privileged code generation path

- Code generator that produces:
  - Readable code
  - Compliant with (railways) standards: limited use of pointers
  - Restrictions on accepted BO (no multi instance)

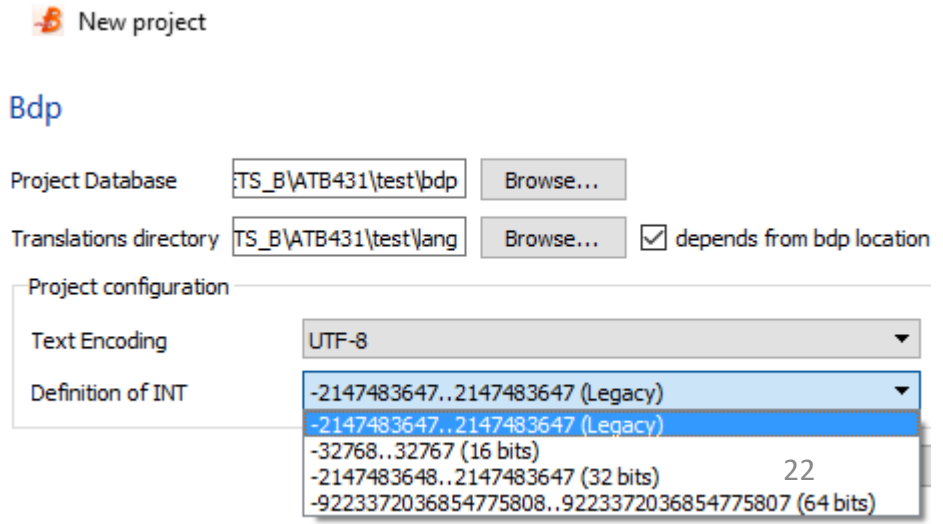
≡ Modes

- Code generation for a:
  - Machine: skeleton for the component
  - Implementation: code for the component
  - Project: makefile and code for the components
- Profiles:
  - Code generation is highly dependant on the target
  - Predefined translation profiles allow adapt generated code



≡ INT

- INT can be redefined at project level to adapt to algorithm precision



C

≡ Applications

- C code embedded at Sao Paulo L15 and Stockholm Citybanan

≡ Embedded systems

- Aimed at low cost hardware (from 6€ to 100€)
- Resources available in the future on github: Training Resources for the B Method

MicroChip PIC32 | Pine 64 | Parallella | Raspeberry Pi | CHIP

Ladder

≡ Programming PLCs

- One language (out of 5) for programming PLCs (IEC61131-3)
- Very Similar to electric circuits

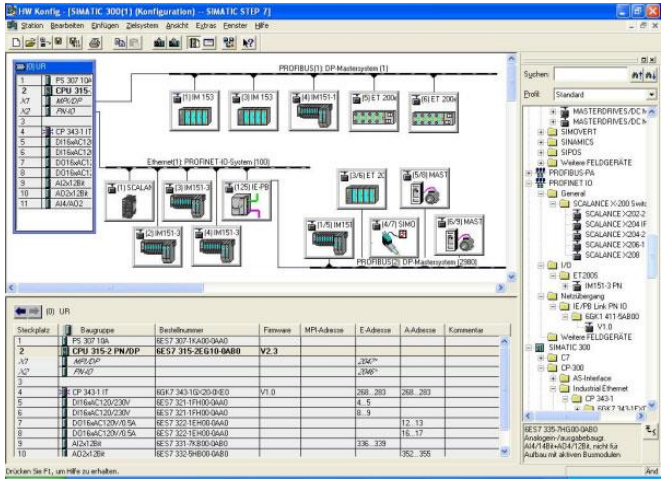
```
1. ----[ ]---------|--[ ]--|------( )--
          X        |   Y   |        S
                   |       |
                   |--[ ]--|
                       Z
```

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

≡ Mandatory IDE

- Step7 IDE required to program Siemens-S7 to keep the SIL3 certificate
- Ladder code generator is not connected to Step7
- Ladder programs have to be manually entered into Step7 with the GUI
- Cross-read required to check identity between programs

- Solution abandonned
- Translator not published because too specific



Instruction List

≡ Programming PLCs

- One language (out of 5) for programming PLCs (IEC61131-3)
- Kind of assembly language for PLCs

≡ Mandatory IDE

- Step7 IDE accept Instruction List copy/paste actions
- Simpler, safer, no further cross-verification

- Translator not published because too specific

≡ Applications

- PSD controller Paris Line 1 (2015), Caracas L2 (2016)
- Bridge Crane Lyon (2013)

```
IMPLEMENTATION  T003_set_i

REFINES  T_empty_mch

LOCAL_OPERATIONS
    bb <-- ope_01(nn) =
    PRE
        bb : BOOL &
        nn : NAT
    THEN
        skip
    END

OPERATIONS
    bb <-- ope_01 (nn) =
    BEGIN
        IF nn=3
        THEN
            bb := FALSE
        END
    END

END
```

```
VAR_INPUT
      nn : INT;
END_VAR

VAR_OUTPUT
      bb : BOOL;
END_VAR

VAR_TEMP
END_VAR

BEGIN

NETWORK
TITLE = RESEAU_1
      U(;
      L       #nn;
      L       3;
      ==I     ;
      );
      NOT     ;
      SAVE    ;
      BEB     ;

NETWORK
TITLE = RESEAU_2
      U       "F_GLOBDB".VKE0;
      =       #bb;

END_FUNCTION
```

Instruction List

Bridge cranes (SNCF Lyon)

- Safe movements over catenary lines

Ground PLC

Crane PLC

Rationale for Using B

B is a language for proving, not for programming

Something needs to be proved

- If your requirements are only "do A_1 then A_2 then A_3", do not use B

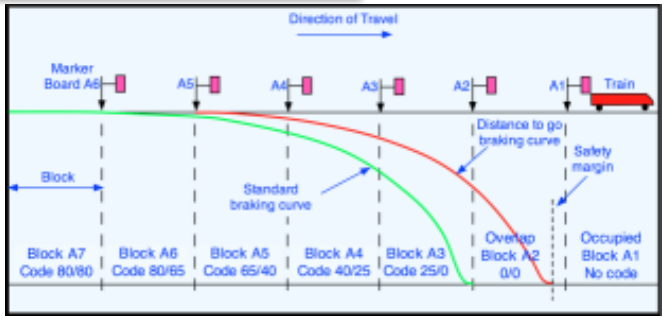
Constraints

- Operation calls of imported / sees MACHINES
- Loops should terminate
  - global while(true) can be proved or should call B operations (sequencer)

B for Safety Critical Railway Applications

≡ Reduce intervals between trains (from 120s to 90s / 75s)

- Passive security not sufficient (power off)
- Active security is required (trains have to brake when emergency)



Braking curves

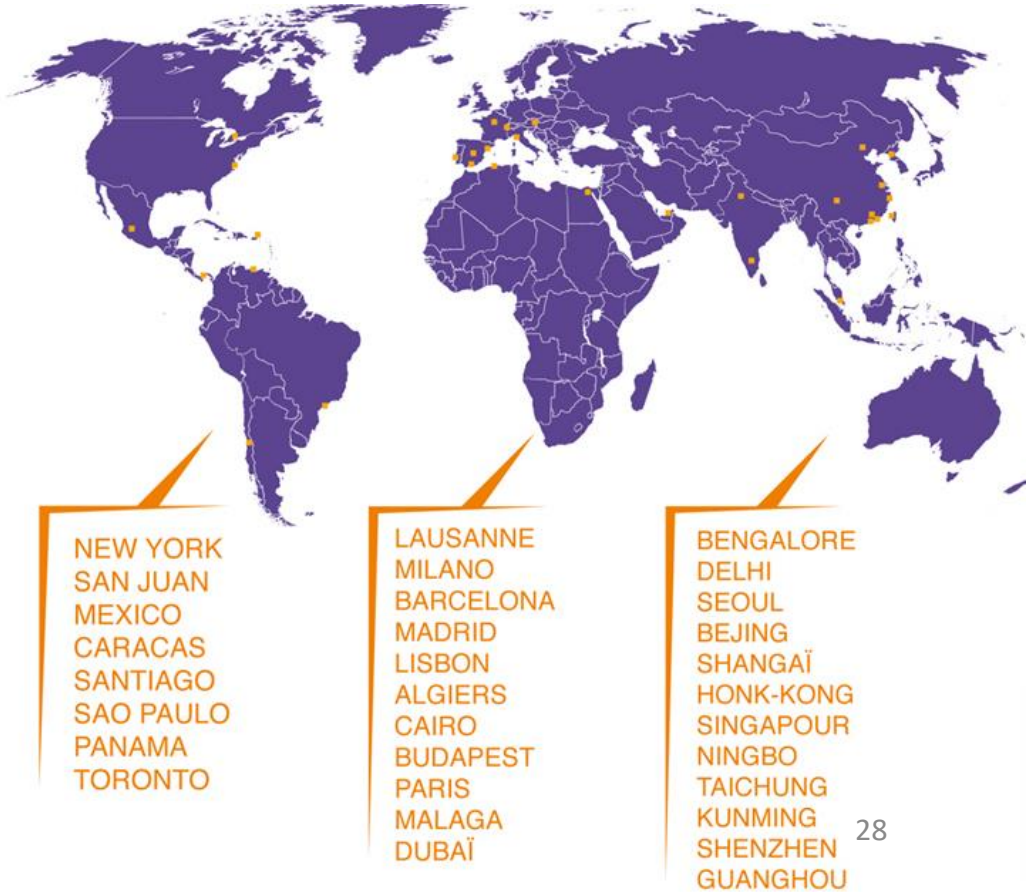
≡ SACEM ATP

- Analyzed with Z: 20 catastrophic failing scenarios found
- Massive public investment for strengthening Alstom CASE tool
- Atelier B ready for action

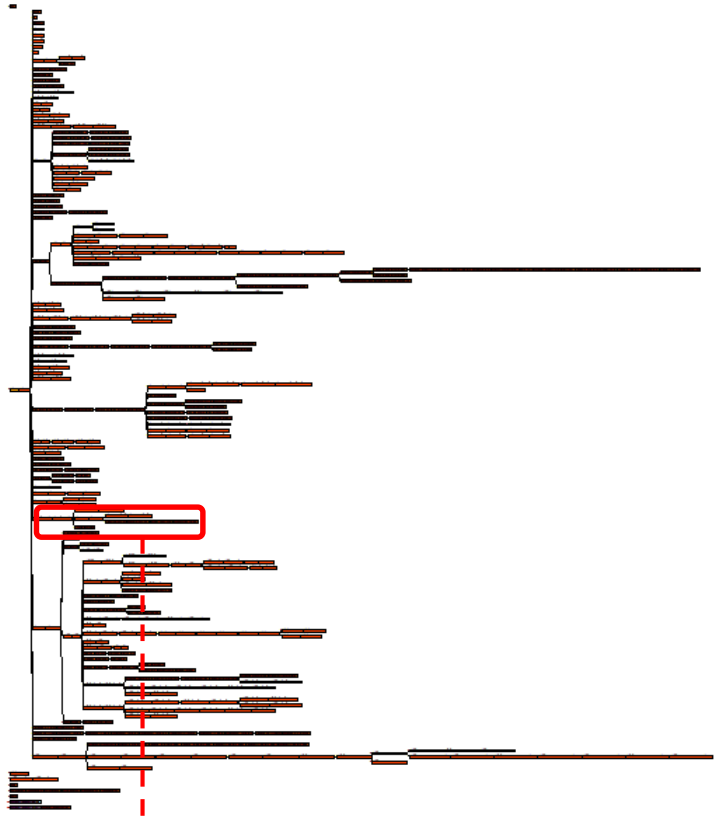
≡ B instead of program proof for

- Embedded software (Automatic Train Pilot)
- Localization: graph-based algorithms
- Energy control: integer arithmetic (braking curve)
- Emergency braking: Boolean predicates
- Trackside software (Interlocking)

≡ +30% automatic metros in the world



Safety Critical Railway Applications



Modern Automatic Train Protection Software (2015)

Top level implementation

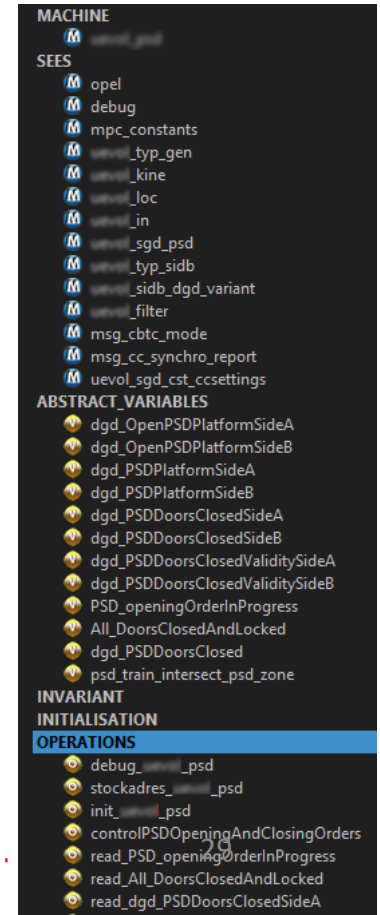
- Imports 55 components
- Specify top level one-cycle function:
  - Compute location, manage kinetic energy, control PSD, trigger emergency braking, etc.

Metrics

- 233 machines, 50 kloc
- 46 refinements, 6 kloc
- 213 implementations, 45 kloc
- 3 000 definitions
- 23 000 proof obligations (83 % automatic proof)
- 3 000 added user rules (85 % automatic proof)

Platform Screen Doors

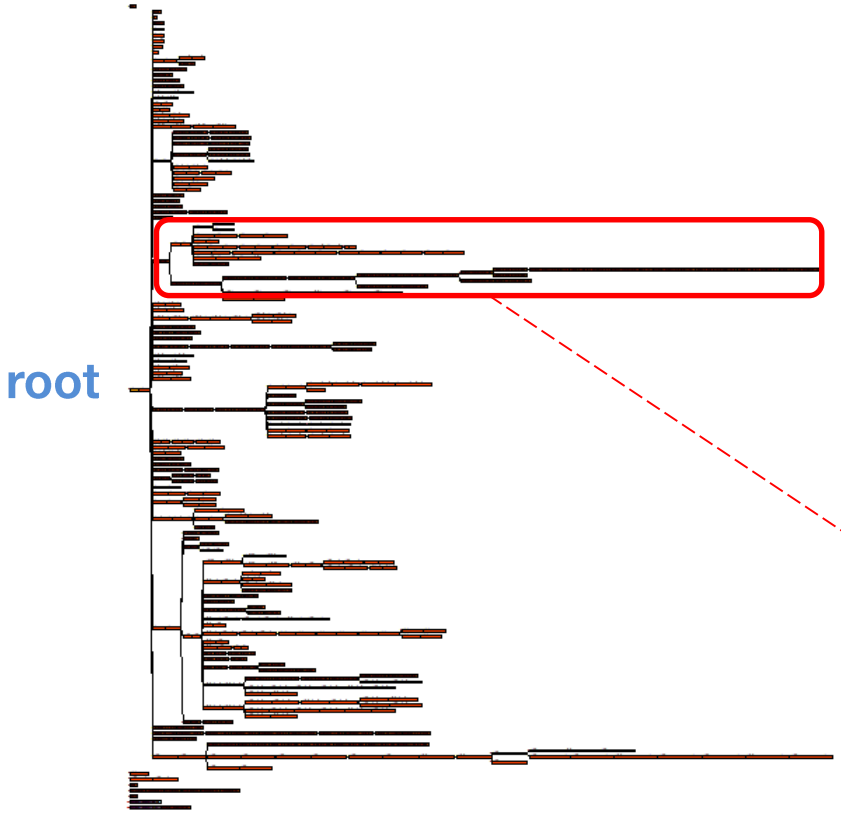
- Top component: 12 variables 14 operations
- 10 components, 2 kloc specification, 2kloc implementation
- Connection with I/O through basic machines



Safety Critical Railway Applications

Biggest function: Localization

where is the train ?



Modern Automatic Train Protection Software (2015)

Post-condition of one operation:

variables become such as …

```
variables :(types &
    properties &
    properties_train &

    (loc_trainLocated = TRUE =>
        0 <= loc_locationUncertainty
        & kine_kineInvalid = FALSE
        & loc_train_track /= {}
        & first(loc_train_track) = loc_ext2Block |-> oppositeDirection(loc_ext2Dir)
        & !ii.(ii : 1..size(loc_train_track)-1 => loc_train_track(ii) : dom(sidb_nextBlock))
        & !ii.(ii : 1..size(loc_train_track)-1 => sidb_nextBlock (loc_train_track(ii)) = loc_
        & #aa.(aa : 1..size(loc_train_track) & prj1(t_block,t_direction)(loc_train_track(aa))


        & loc_rearBlock = { c_cabin1 |-> loc_ext2Block, c_cabin2 |-> loc_ext1Block, c_none |-
        & (loc_rearBlock = c_block_init
            =>loc_rearDir=c_up)
        & ( not (loc_rearBlock = c_block_init)
            =>  loc_rearDir    = { c_cabin1 |-> oppositeDirection(loc_ext2Dir), c_cabin2 |-> c
        & loc_rearAbs    = { c_cabin1 |-> loc_ext2Abs, c_cabin2 |-> loc_ext1Abs, c_none |-> 0
        & loc_frontBlock = { c_cabin1 |-> loc_ext1Block, c_cabin2 |-> loc_ext2Block, c_none |
        & loc_frontDir   = { c_cabin1 |-> loc_ext1Dir, c_cabin2 |-> loc_ext2Dir, c_none |-> c
        & loc_frontAbs   = { c_cabin1 |-> loc_ext1Abs, c_cabin2 |-> loc_ext2Abs, c_none |-> s
    ))
```

« improved development cycle »

« Only inactive sequences can be added to the active sequences execution queue. »

Natural language requirement

```
activation_sequence = /* Activation d'une séquence non active */
PRE ¬(sequences = sequences_actives) THEN
    ANY sequ WHERE
        sequ ∈ sequences - sequences_actives
    THEN
        sequences_actives := sequences_actives ∪ {sequ}
    END
END;
```

B Specification

```
activation_sequence = /* Activation d'une séquence non active */
VAR sequ IN
    sequ <-- indexSequenceInactive;
    activeSequence(sequ)
END;
```

B Implementation

```
void M0__activation_sequence(void)
{
    CTX__SEQUENCES sequ;

    sequence_manager__indexSequenceInactive(&sequ);
    sequence_manager__activeSequence(sequ);
}
```

C generated code

```
0x01F970 FFFF 8B4C 2440 89C5 8D7D 0C8B 4110 89CE
0x01F980 83C6 0C8D 1485 0000 0000 8D42 0883 F807
0x01F990 7617 F7C7 0400 0000 740F 8B41 0C8D 7D10
0x01F9A0 83C6 0489 450C 8D42 04FC 89C1 C1E9 02F3
```

Binary code

- Specification and design coding under control
- Usually more time spent during specification
- V development cycle is mainly \ development cycle

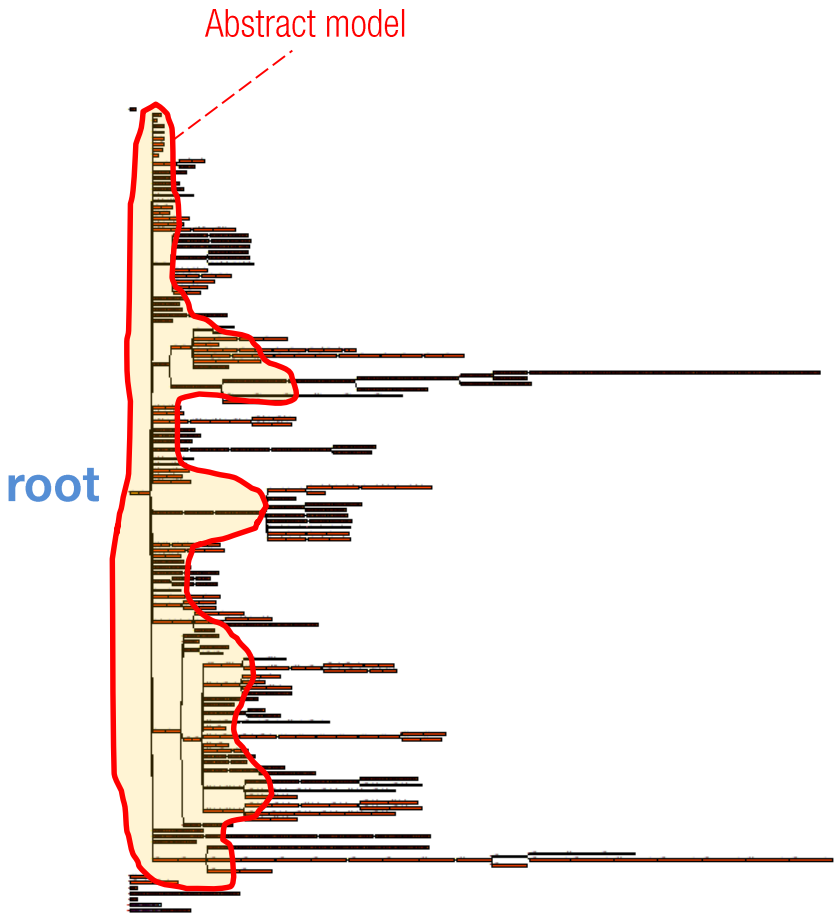
- However potential problems at interfaces
  - Are requirements captured by formal specification ?
  - Is generated source code correct ?
  - Is binary code correct ?

Improved « improved development cycle »

Several techniques to speed up convergence:

- Automatic refinement
- Traceability
- Model animation
- Simplified abstract modelling

Automatic Refinement

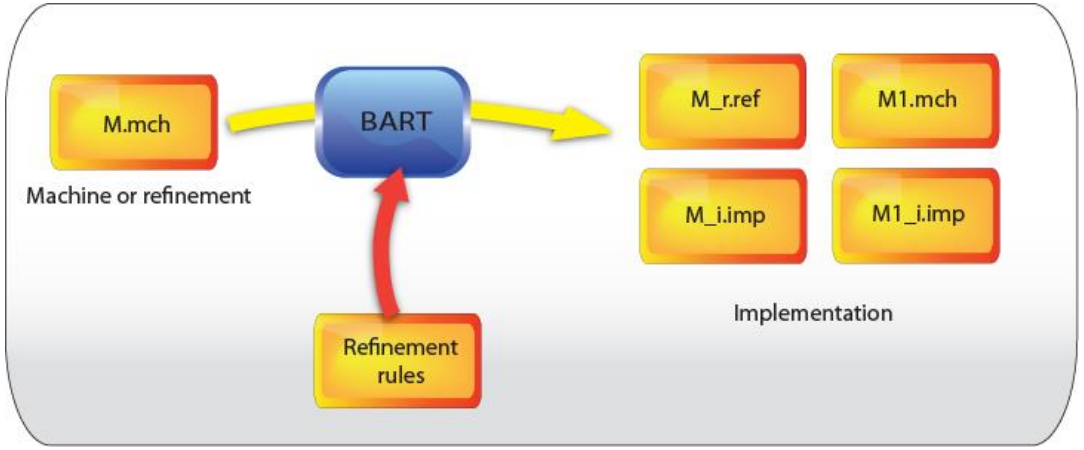


Modern Automatic Train Protection Software (2015)

≡ Progressive transformation of a model containing all design decisions into a model able to be translated into an imperative language

≡ Initially invented by Matra Transport

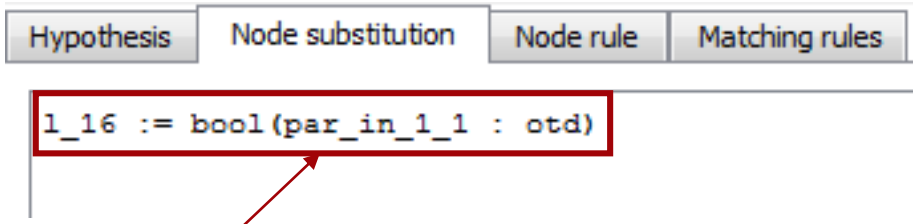
- To support their own development methodology (abstract model first)
- In-house tools (EdithB + Bertille) to obtain an efficient process
- Refinement engine based on refinement rules



Automatic Refinement, L. Burdy, J.M. Meynadier, FM 1999
Return of Experience on Automated Refinement in B, T. Lecomte, SETS 2014

Refinement Rules & Pattern Matching

Substitution to refine



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)
```

From B Specification to B Implementation

Human

```
OPERATIONS
    control =
    BEGIN
        s0 := bool(TRUE: {e0, e1, e2})
    END
END
```

M0

M0_i

M0_1

M0_1_i

Automatically generated (refined)

Refining tool

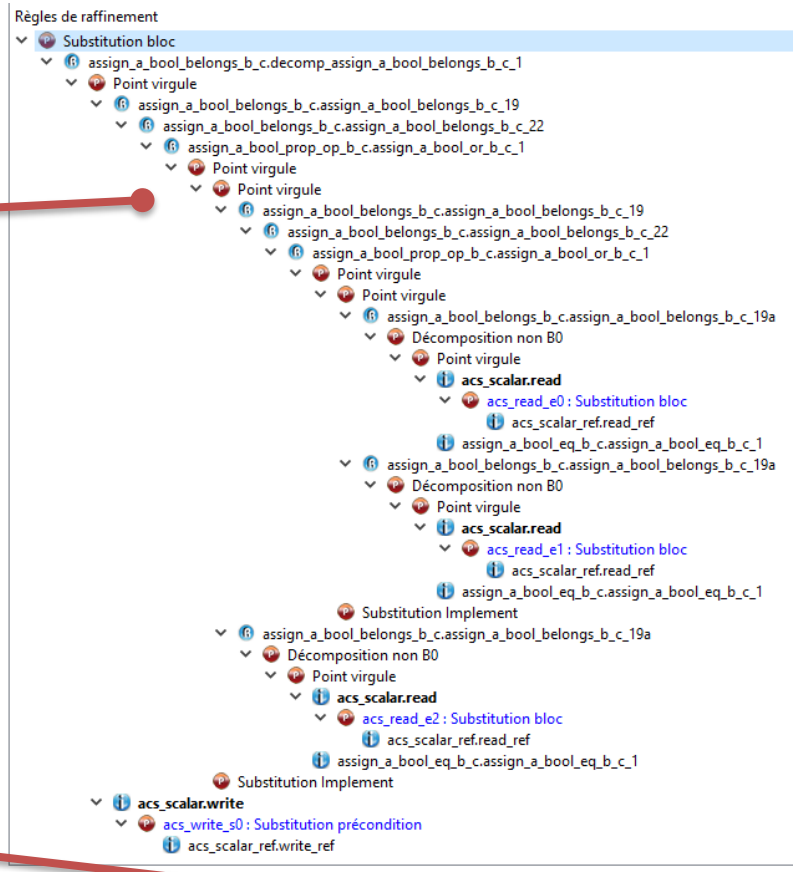
```
control =
VAR
  l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8
IN
  /* Rule : assign_a_bool_belongs_b_c.decomp_assign_a_bool_belongs_b_c_1 */
  /*? l_6 := e0 ?*/
  /* Rule : acs_scalar.read */
  l_6 <-- acs_read_e0 ;
  /* Rule : assign_a_bool_eq_b_c.assign_a_bool_eq_b_c_1 */
  l_4 := bool(TRUE = l_6) ;
  /*? l_7 := e1 ?*/
  /* Rule : acs_scalar.read */
  l_7 <-- acs_read_e1 ;
  /* Rule : assign_a_bool_eq_b_c.assign_a_bool_eq_b_c_1 */
  l_5 := bool(TRUE = l_7) ;
  l_2 := bool(l_4 = TRUE or l_5 = TRUE) ;
  /*? l_8 := e2 ?*/
  /* Rule : acs_scalar.read */
  l_8 <-- acs_read_e2 ;
  /* Rule : assign_a_bool_eq_b_c.assign_a_bool_eq_b_c_1 */
  l_3 := bool(TRUE = l_8) ;
  l_1 := bool(l_2 = TRUE or l_3 = TRUE) ;
  /*? s0 := l_1 ?*/
  /* Rule : acs_scalar.write */
  acs_write_s0(l_1)
END
```

From B Specification to B Implementation

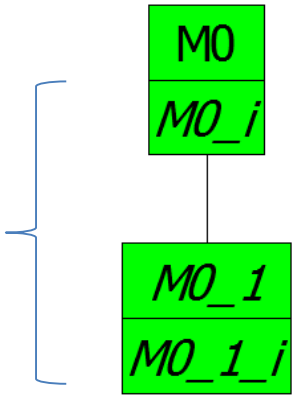
Human

```
OPERATIONS
    control =
    BEGIN
        s0 := bool(TRUE: {e0, e1, e2})
    END
END
```

Refinement tree



Automatically generated (refined



Proof entirely automatic

- So is assigned a boolean
- Operation implements its specification

Composant	Typage vérifié	OPs générées	Obligations de Preuve	Prouvé	Non-prouvé
M0	OK	OK	0	0	0
M0_1	*OK*	*OK*	*0*	*0*	*0*
M0_1_i	*OK*	*OK*	*0*	*0*	*0*
M0_i	*OK*	*OK*	*1*	*1*	*0*

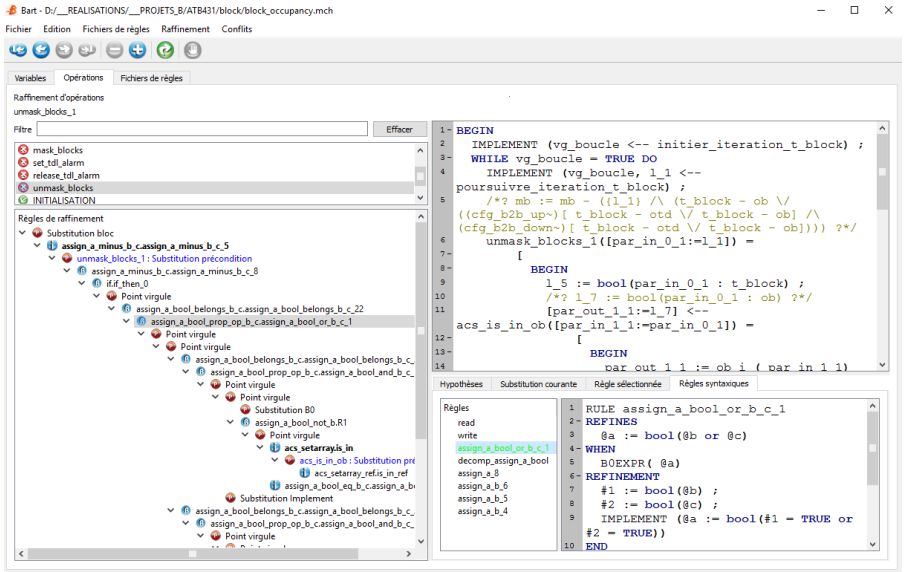
Automatic Refinement

≡ No need to certify the tool

- Whatever the refinement rules, in the engine or added by users
- If the refined model is incorrect, it can't be proved

≡ Impact on projects

- Siemens claimed to divide development costs by 2 when the process is stabilized
- More little steps, more decomposition levels, more function calls, more lines of code
- Refinement patterns ease deployment of proof strategies



B Automatic Refinement Tool (BART) added to Atelier B in 2007, in accordance with Siemens T.S.

≡ Industrial applications

[2004] Canarsie line (ATP): *38k lines of handwriten B, 115k lines of generated B*

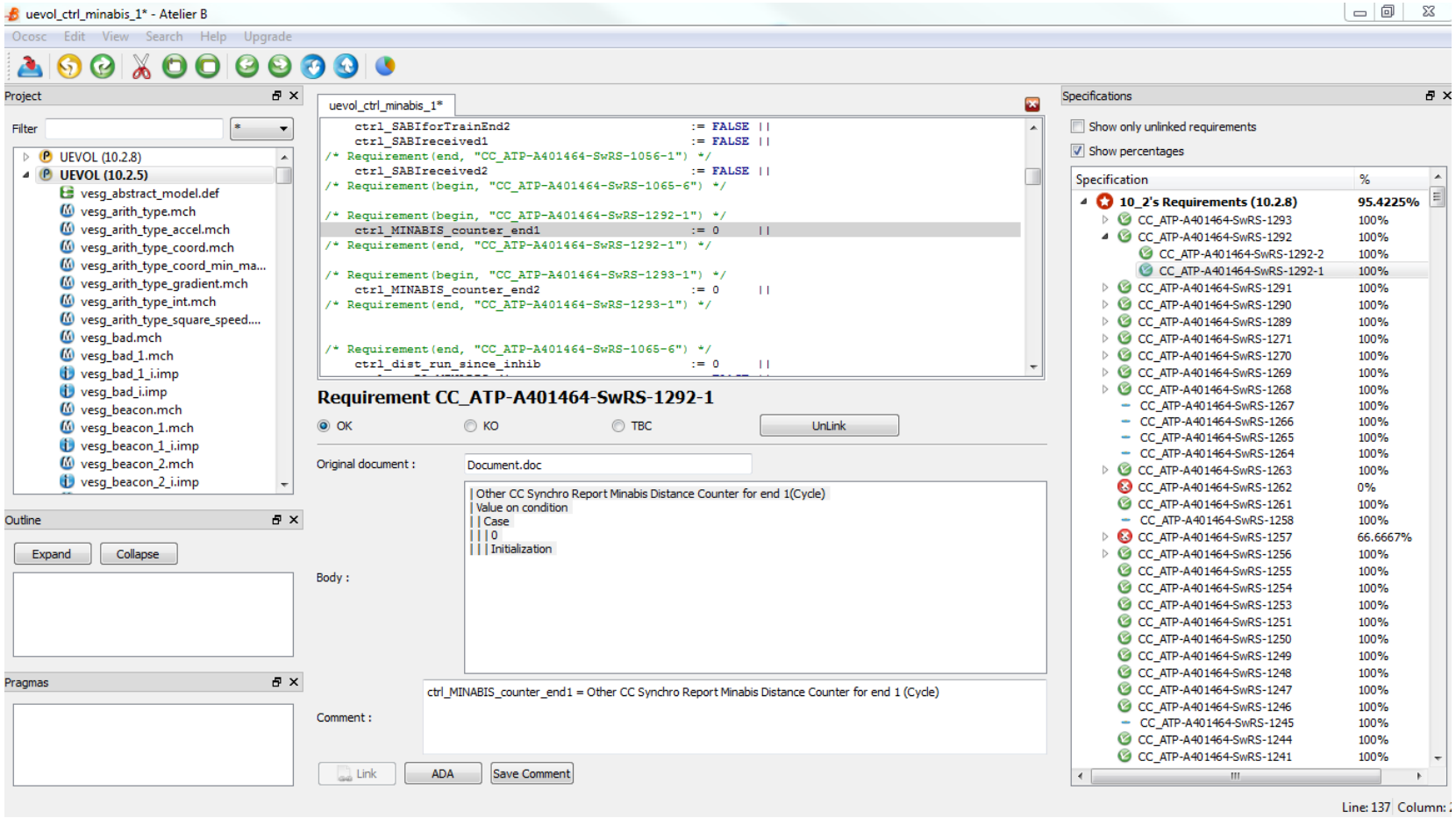
[2007] Roissy airport shuttle (ATP+ZC): *40k lines of handwritten B 225k lines of generated B*

[2007] Symbolic Calculus Engine: *200k lines of generated B*

[2015] SysML Compiler : *300k lines of generated B*

Traceability / Conformance

Dedicated tool, integrated to Atelier B, to ensure traceability between natural language document and B models, over versions.

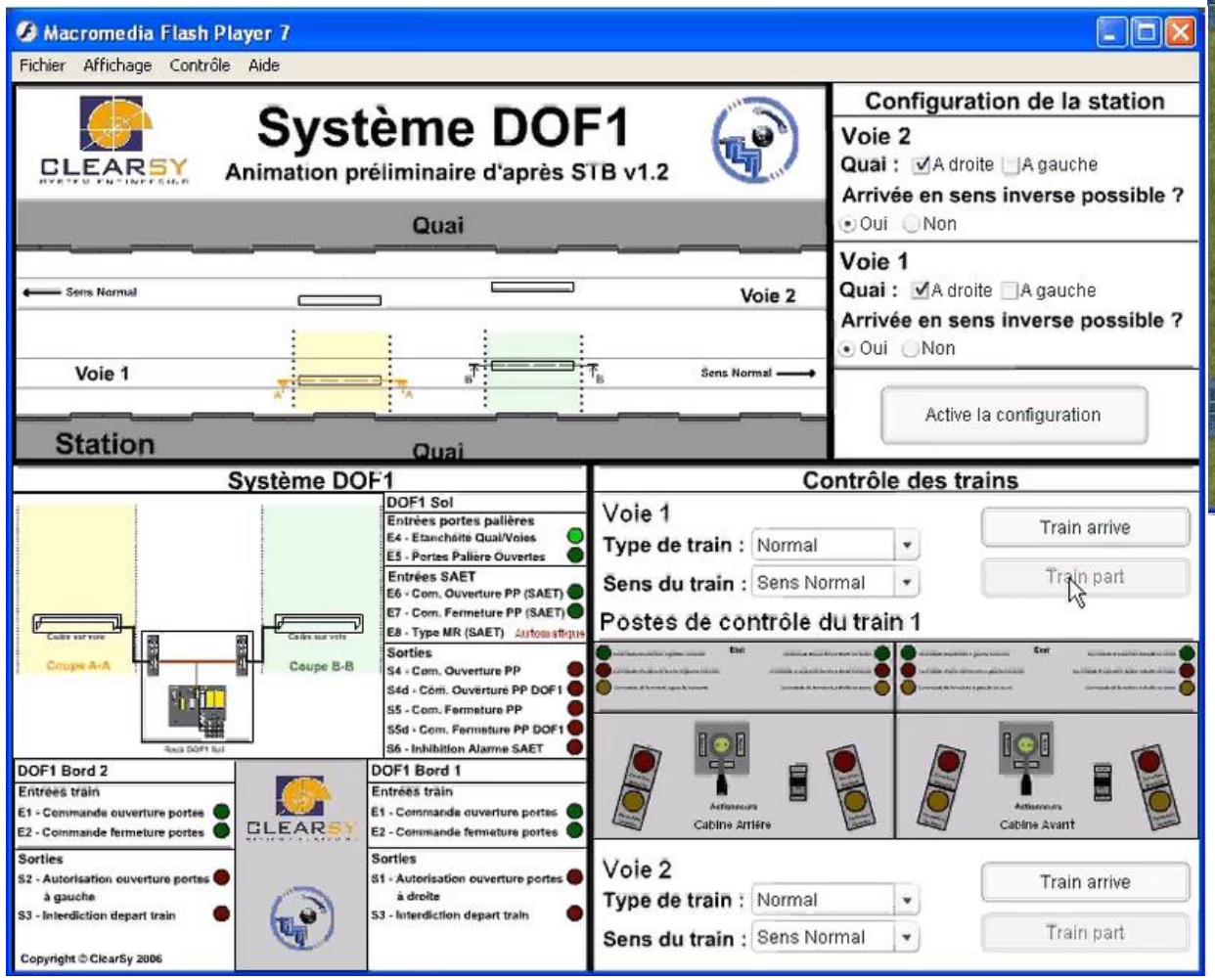


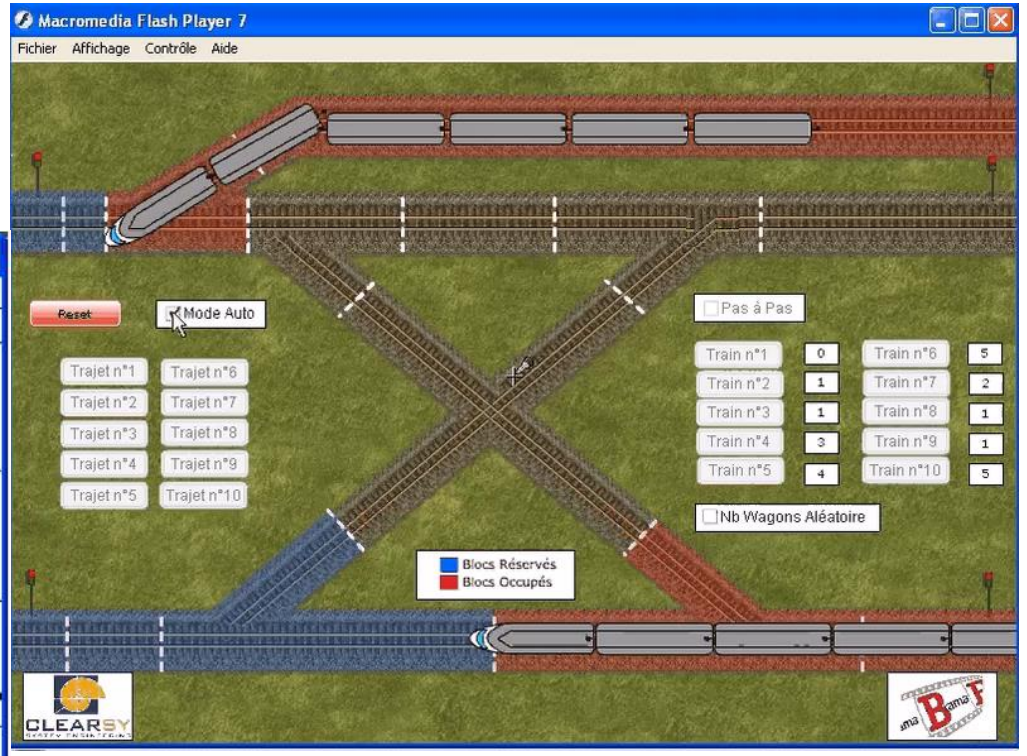
On-going action with LACL to provide a formal framework for requirements modelling and verification, « à la KAOS »

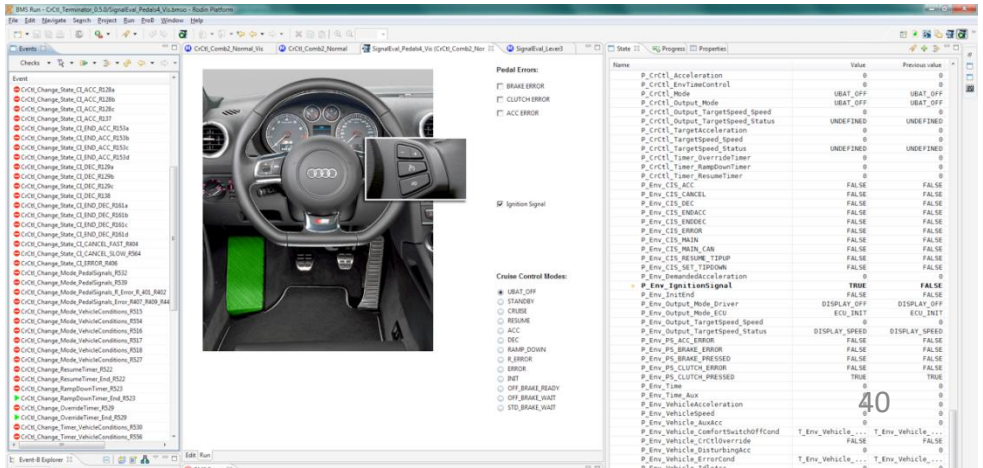
Model Animation

Model animation for:

- Debugging
- Convincing experts/customers



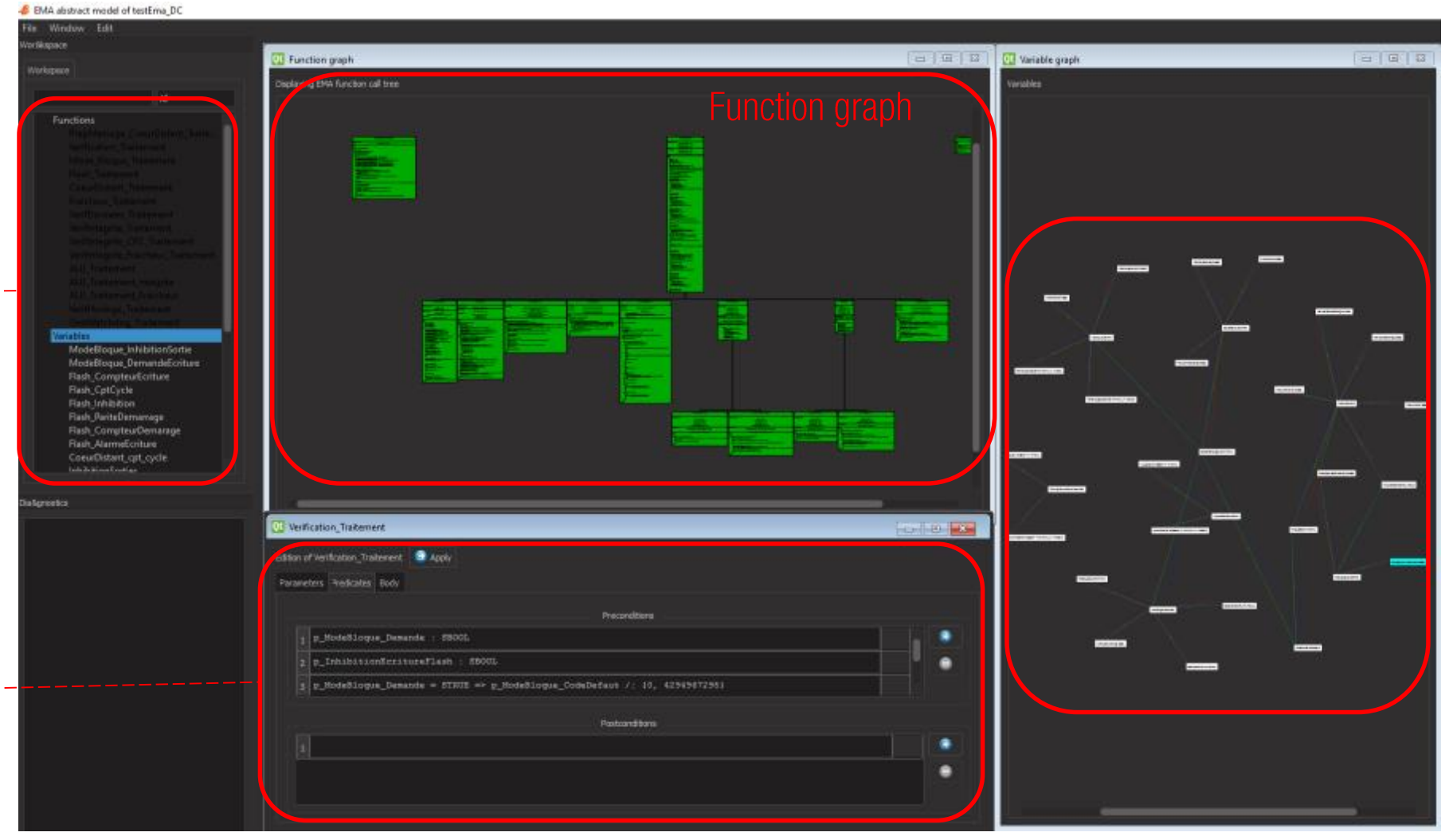




Simplified Abstract Modelling

≡ Propose a new way of modelling

- User is incited to think in terms of functions
- Model creation, decomposition, variables allocation is left to the computer



Outline
- Functions
- Variables
- Constants

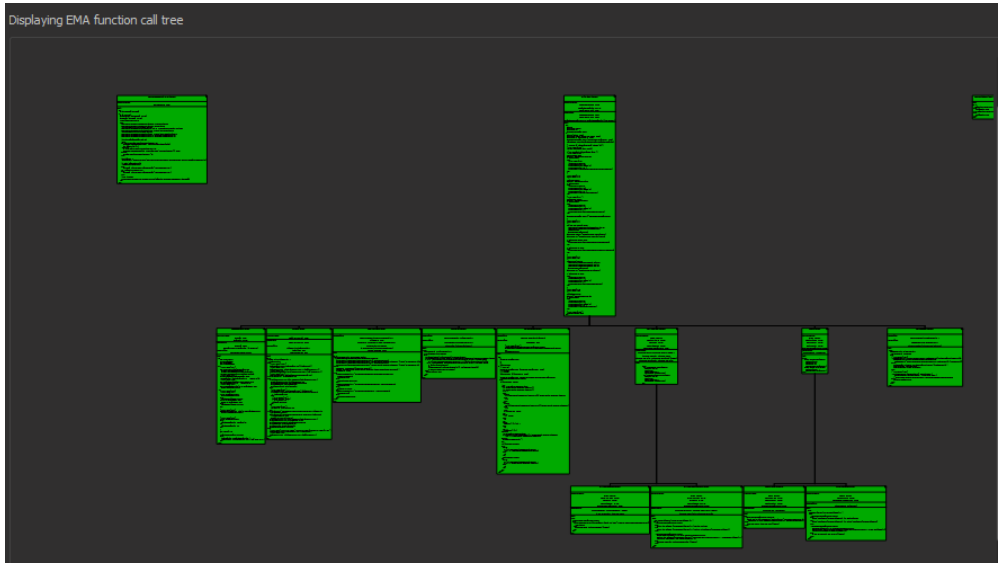
Function
- Parameters
- Predicates
- Body

Variables graph
- What is produced
- What is consumed

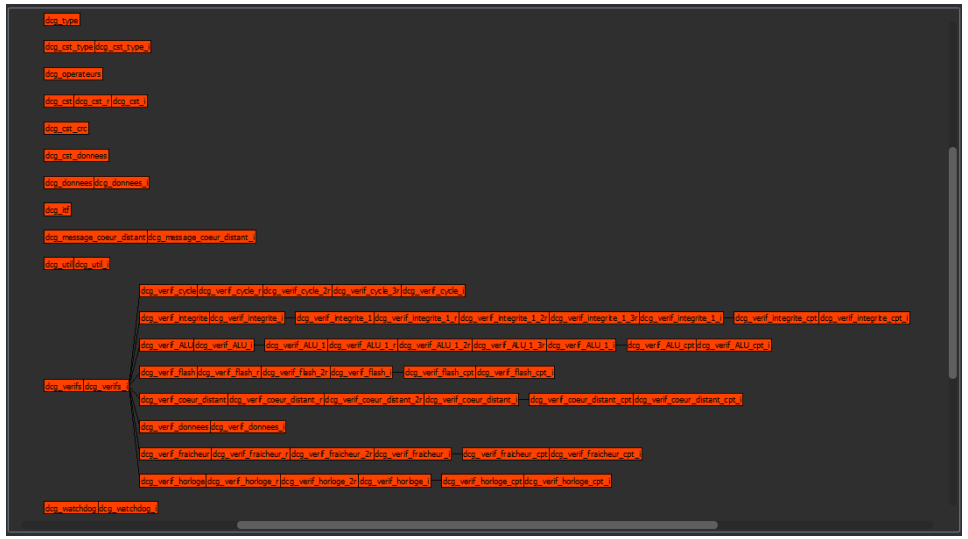
Simplified Abstract Modelling

≡ Two Modelling Views for the Same Project

- Projects can me edited what ever the view
- Providing a clear distinction between abstract model and concrete model
- Corollary: not all projects can be analysed
- Direction continuation of the Automatic Refinement approach



Abstract Modelling Editor (EMA) to appear in Atelier B 4.4



Atelier B current component view

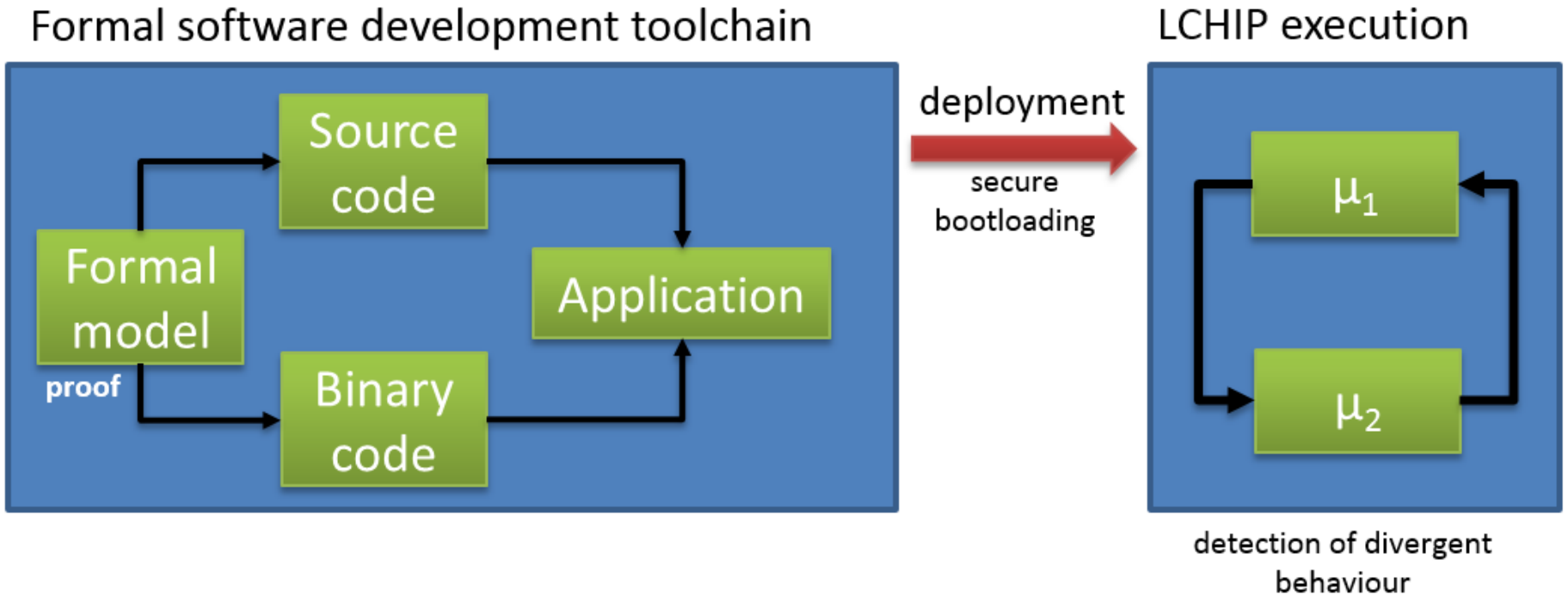
Low Cost High Integrity Platform

Genesis

- **Safety critical applications development**
  - Higher development costs (safety case)
  - Expensive execution platforms
  - Operational constraints
- **Impact**
  - Rigid solutions
  - Prohibitive costs
  - Lack of available of key personal resources

- **Secure architecture**
  - Combination of software and hardware techniques compliant with EN5012{6, 8, 9}
    - Bi-processor using standard hardware
    - Redudant and proved software development process
  - Design of safe (SIL4), low cost automation (+ certification kit)

Principles (existing)



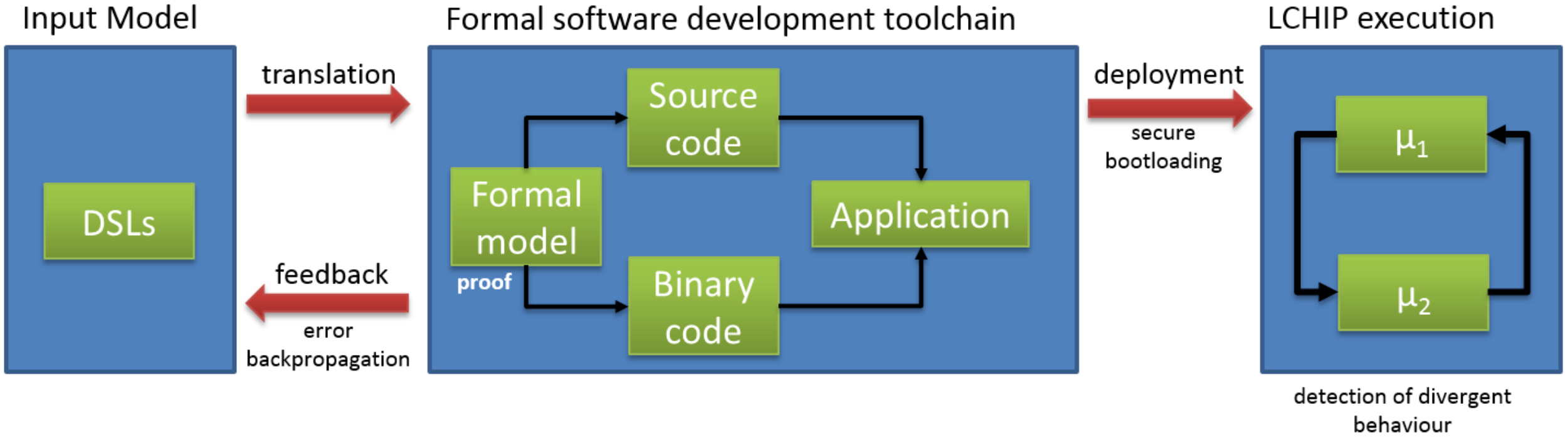
Atelier B
C code generator
Hex compiler
Linker

Unique determinist B model
C code generation and compilation
Hex binary code generation
Production of a unique binary file
Verification of correct loading

Execution on 2x PIC32
100 MIPS usable
RM if default
Instructions testing

PSD Sao Paulo et Stockholm, gateway SATURN
SIL4 Certification (CERTIFER)

Principles (LCHIP)



Translation DSLs to B
Errors backpropagated to DSLs

Unique determinist B model
C code generation and compilation
Hex binary code generation
Production of a unique binary file
Verification of correct loading
Automatic refinement of B models
Automatic proof of B models

Execution on 2x PIC32
100 MIPS usable
RM if default
Instructions testing
Daughter board
SCADA (supervision)

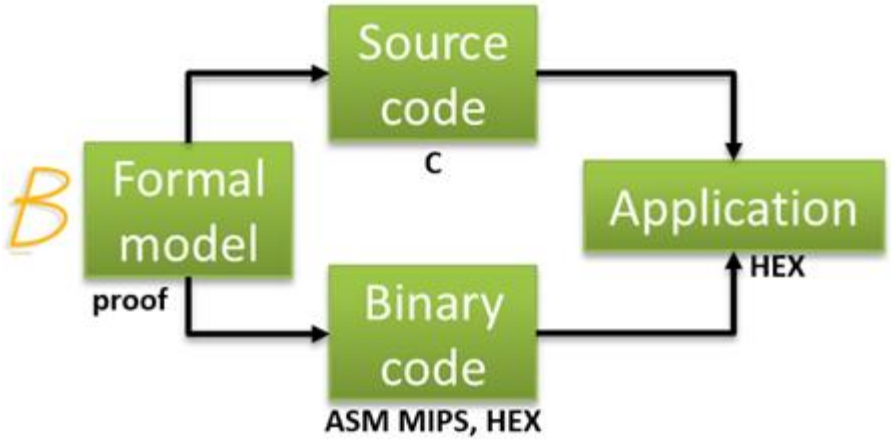
LCHIP
LOW-COST
HIGH-INTEGRITY
PLATFORM

Low Cost High Integrity Platform

≡ 1 B model (application)

- Data
  - Integer arithmetic
  - Boolean equations
  - State machine
  - Variables typing and properties
- Cyclic software, no interrupt modifying state variables

≡ 1 C code generator + commercial compiler => Hex



≡ Application proved vs requirements

- Implementation doesn't contradict specification
- A minima, no programming error:
  - overflow,
  - div 0,
  - tables

≡ 1 ASM MIPS / Hex code generator

≡ Low level software ensuring safety developped in B

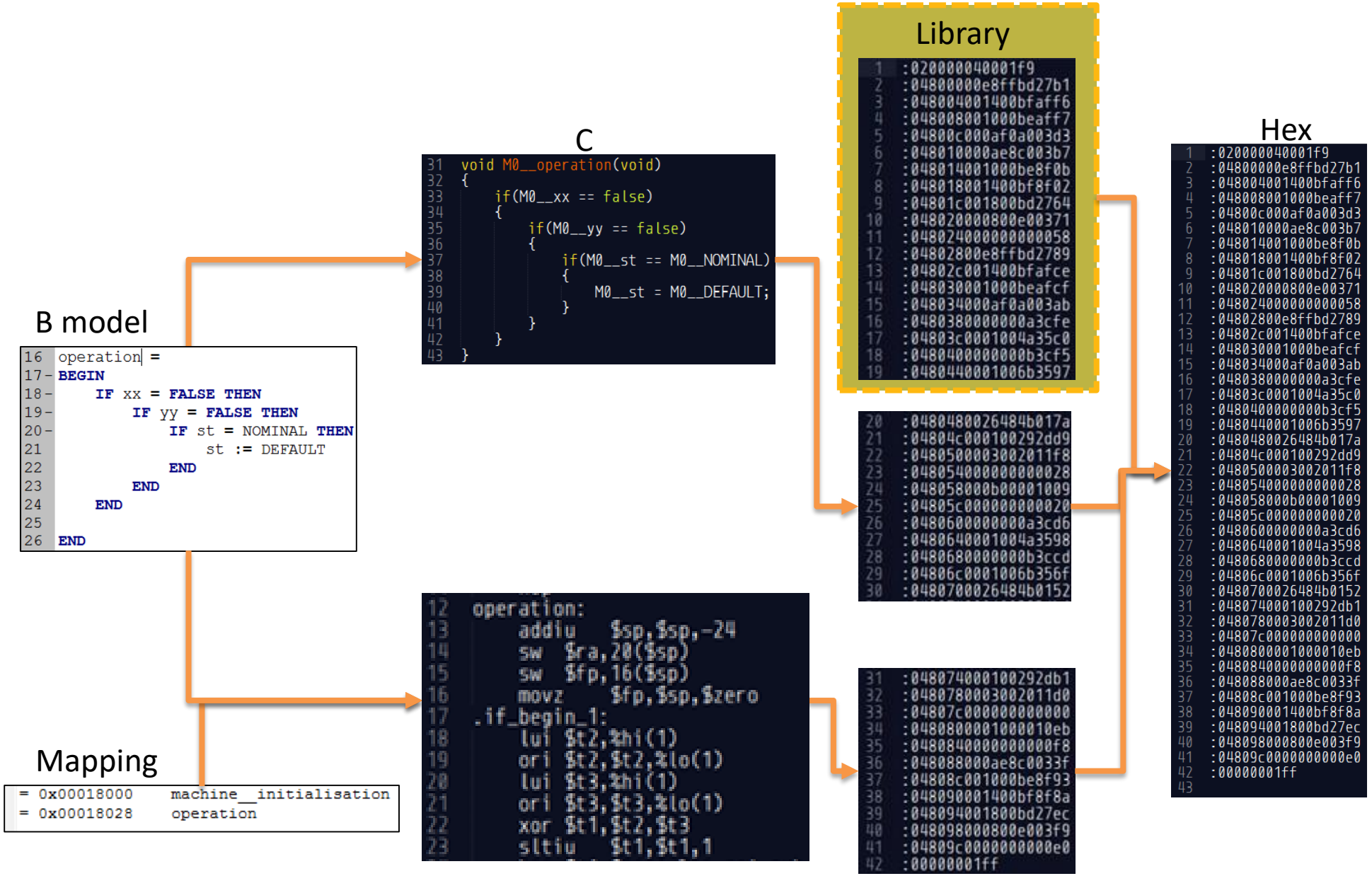
- Once for all
- Safety features out of reach of the developers
- No requirement to certify the tools

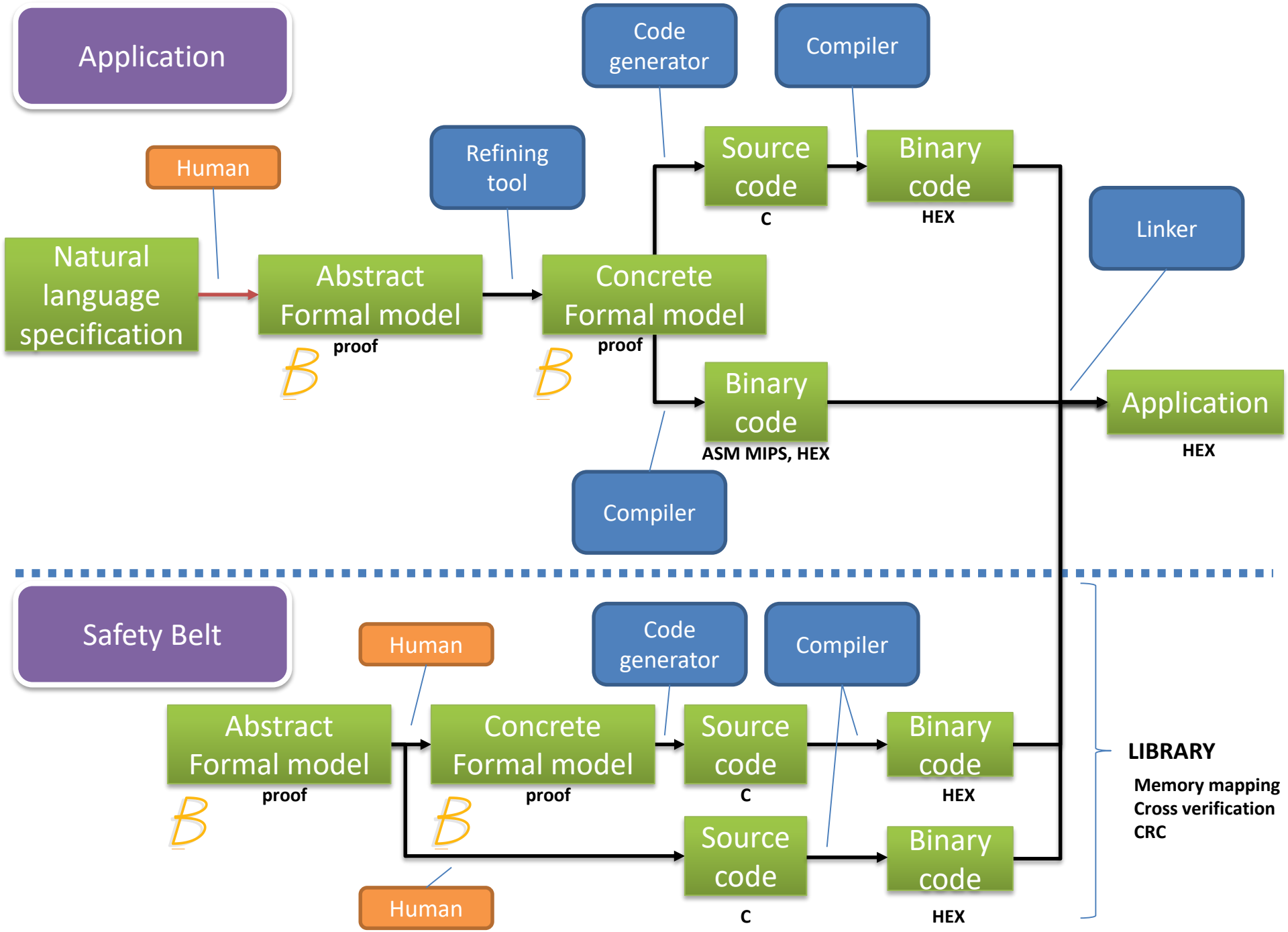
≡ Safety related features

- Instruction testing: instructions are tested at each iteration (fully functional microprocessor)
- Performances are checked regularly (counters) in case one microprocessor is running too slow
- Variables locations checked when loading applications (distinct variables, enough memory)

Funded by bpifrance

Binary


Library
C
Hex
B model
Mapping
operation =
BEGIN
IF xx = FALSE THEN
IF yy = FALSE THEN
IF st = NOMINAL THEN
st := DEFAULT
END
END
END
END
= 0x00018000 machine__initialisation
= 0x00018028 operation


Application
Code generator
Compiler
Human
Refining tool
Source code
C
Binary code
HEX
Linker
Natural language specification
Abstract Formal model
proof
Concrete Formal model
proof
Binary code
ASM MIPS, HEX
Application
HEX
Compiler
Safety Belt
Human
Code generator
Compiler
Abstract Formal model
proof
Concrete Formal model
proof
Source code
C
Binary code
HEX
Source code
C
Binary code
HEX
Human
LIBRARY
Memory mapping
Cross verification
CRC

Low Cost High Integrity Platform

≡ Starter-kit @ Q3 2017

- Mother board (I/Os, maintenance processor, network sockets)
- Daughter card, embedding the double-processor
- IDE v1, based on Atelier B

≡ IDE v2 @ Q2 2018

≡ IDE v3 @ Q2 2019

Conclusion

≡ B is an efficient way of handling safety critical software when properties are identified

- Problems are detected while software is modelled
- Not when software is complete
  - testing,
  - program proof with assertions (Frama-C)

≡ B doesn't require PhD-only teams

- But one guru and engineers able to asbtract « things »

≡ Innovations in

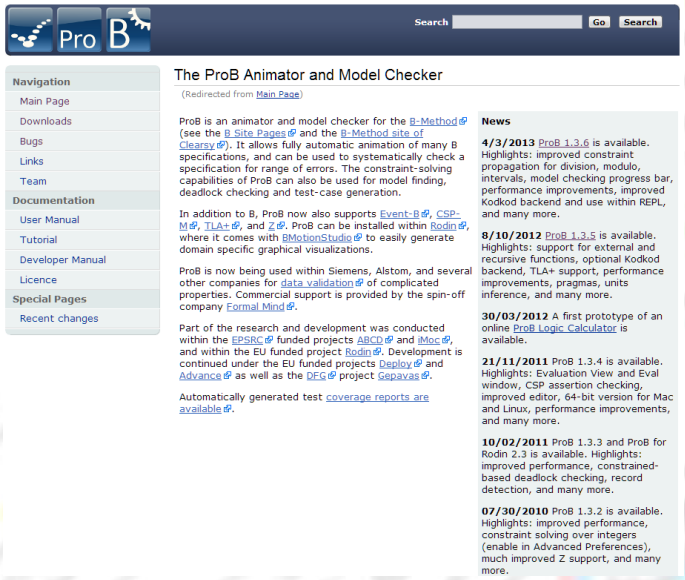
- Proof -> mostly automatic on « not too complicated » predicates (Alt-Ergo, ProB, etc.)
- Safety hardware based on low cost computers (hardware costs divided by an order of magnitude)

make B elligible for other application domains (energy/nuclear in particular)

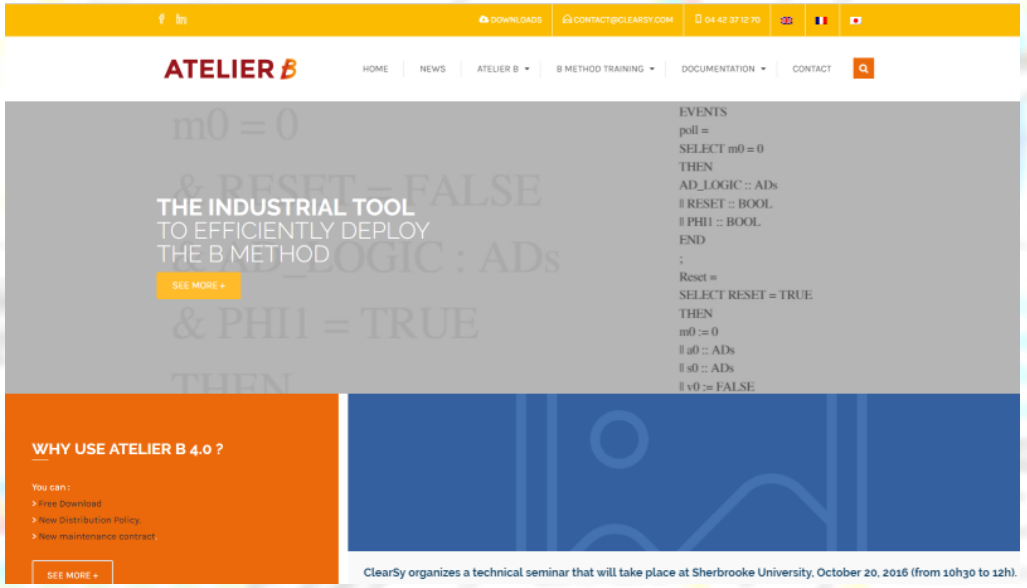
≡ Event-B

- System level models and proofs
- Smartcard certification
- Safety assessment of large systems (metro lines, military vehicles)

Thank you for your attention



http://www.stups.uni-duesseldorf.de/ProB



www.atelierb.eu