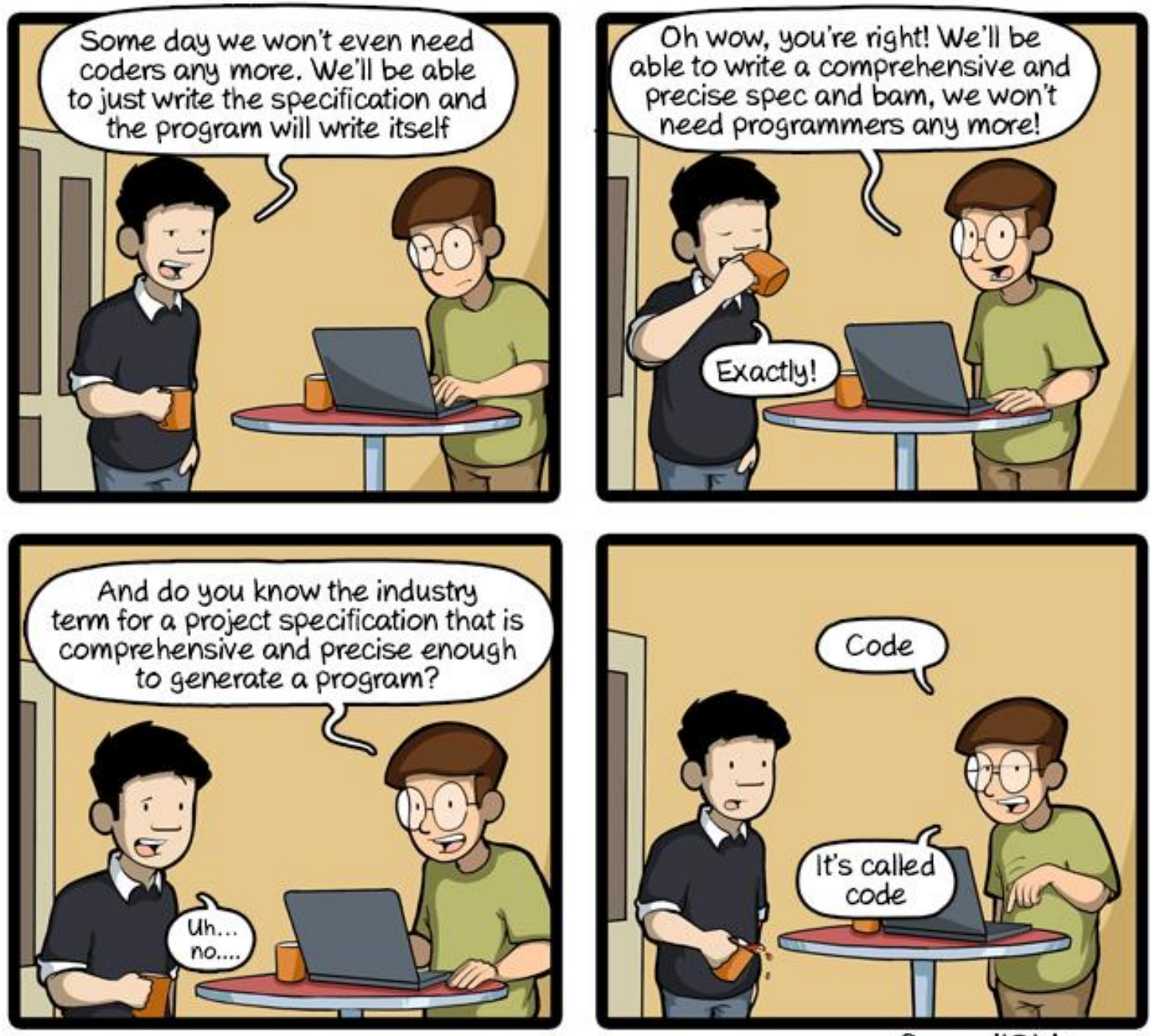
Safety Critical Applications Development with Atelier B

Thierry Lecomte
ETMF 2016
Natal

CLEARSY
SYSTEM ENGINEERING


Some day we won't even need coders any more. We'll be able to just write the specification and the program will write itself
Oh wow, you're right! We'll be able to write a comprehensive and precise spec and bam, we won't need programmers any more!
Exactly!
And do you know the industry term for a project specification that is comprehensive and precise enough to generate a program?
Uh... no....
Code
It's called code
CommitStrip.com

Agenda

Introduction

- Safety critical
- B in a nusthell

A quick example covering the whole process

More complex examples

- Specification
- Refinement
- Implementation
- Architecture
- Proof
- Code generation

Introduction

- Safety Critical
- B in a nutshell

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

Introduction - Safety Critical Systems

≡ Standards

• 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

Highest level

Introduction - Safety Critical Systems

≡ Specific architecture:

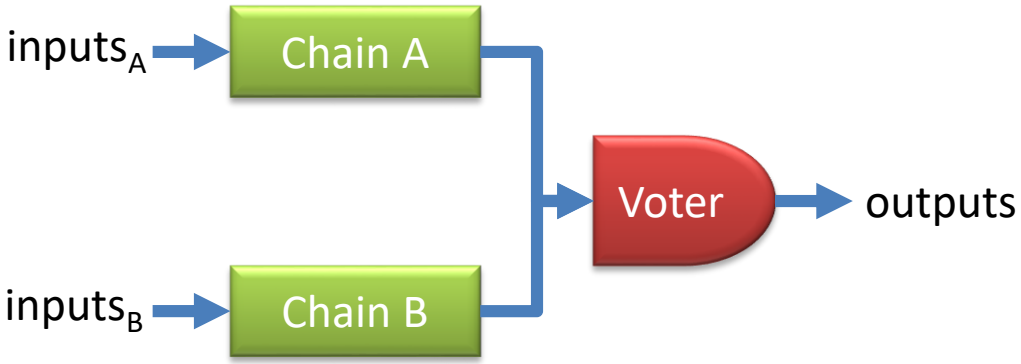
SIL3: 10^{-7} failure / hour

SIL4: 10^{-9} failure / hour

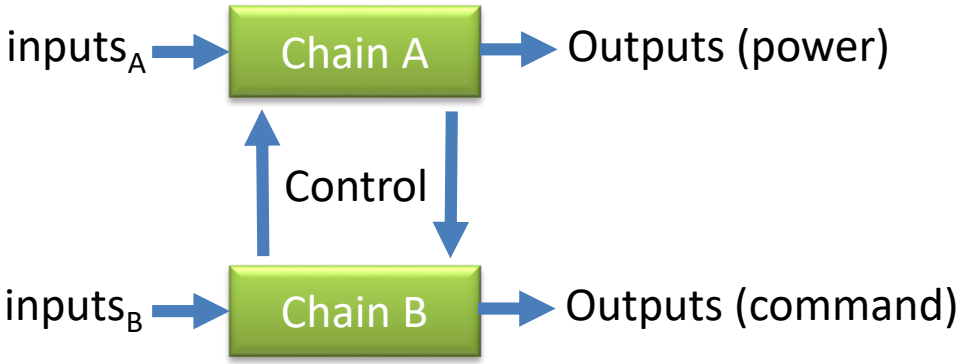
µprocessor: 10^{-5} failure / hour

SIL3, SIL4: 2 µprocessors required

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



Example: 2oo2 architecture with voter

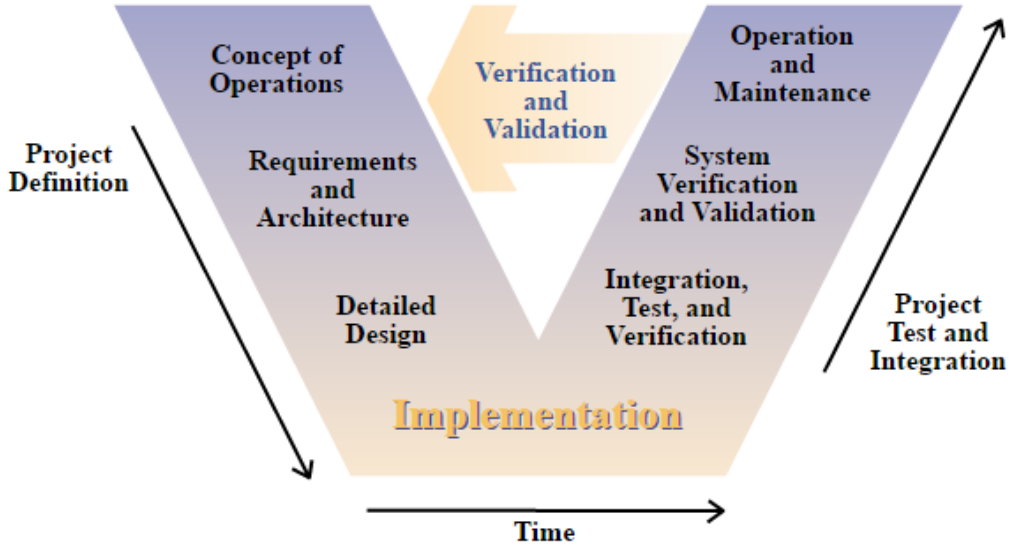


Example: 2oo2 architecture with cross-verification

Introduction - Safety Critical Software

≡ Specific organisation:

- Development
- Verification & Validation (V&V)
- Safety



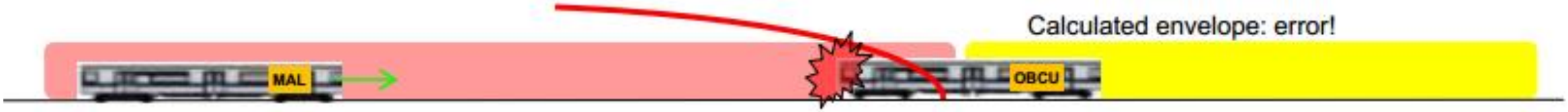
≡ Specific development:

- Safety critical software costs = **2x** Non-Safety critical software costs
- More time spent at specification level, testing, validating
- SIL3-SIL4: 2 software developed by two different teams by using different technologies (avoid common mode failure)

Introduction - Safety Critical Railway Applications

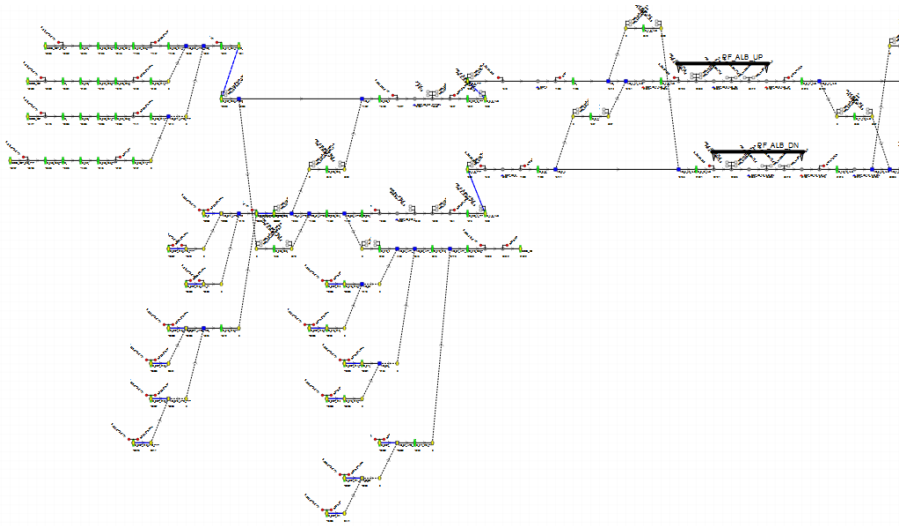
≡ Reduce intervals between trains (from 120s to 90s / 75s)

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



≡ Safety critical functions

- Automatic Train Pilot
- Localization of the train on the track (graph-based algorithms)
- Energy control: computation of braking curve (integer arithmetic)
- Emergency braking (Boolean predicates)
- Interlocking: allocation of tracks to routes



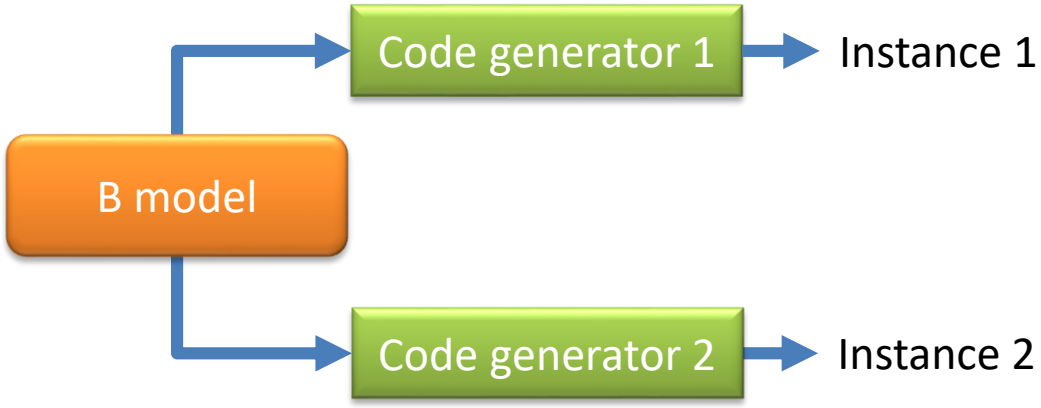
Introduction - Safety Critical Railway Applications

≡ B for software development of safety critical functions

- First automatic metro: Paris L14 in 1998

≡ +30% automatic metros in the world

≡ No need to have two dev teams



Introduction

- Safety Critical
- B in a nutshell

A path from requirements to code

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

```
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;
```

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

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

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

```
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
```

Natural language requirement

Behaviour + properties

B Specification

Behaviour + properties

B Implementation

C generated code

Binary code

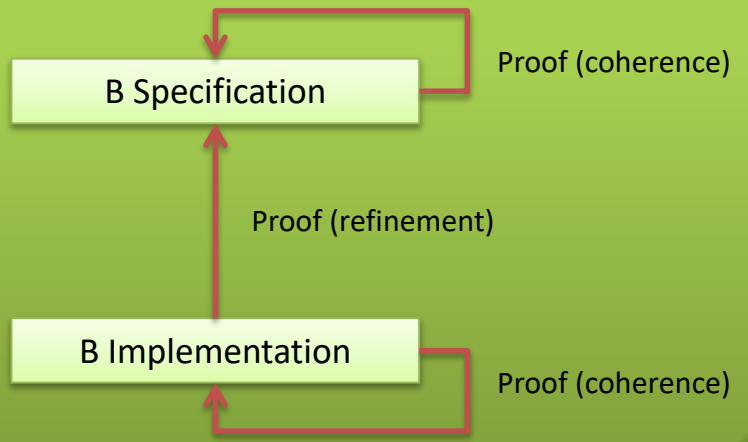
B in a nutshell

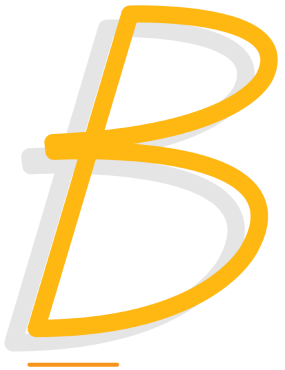
« Only inactive sequences can be added to the active sequences execution queue. »

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;
```

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





```
void M
{
    CT

    se
    se
}
```

Philosophy:
Avoid to introduce errors during the development (proof)
instead of trying to detect them close to the end of the development (tests)

```
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

B (101)

- formal method (semantics)
  - text-based models of unthreaded programs, no interruption modifying the state variables
  - 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

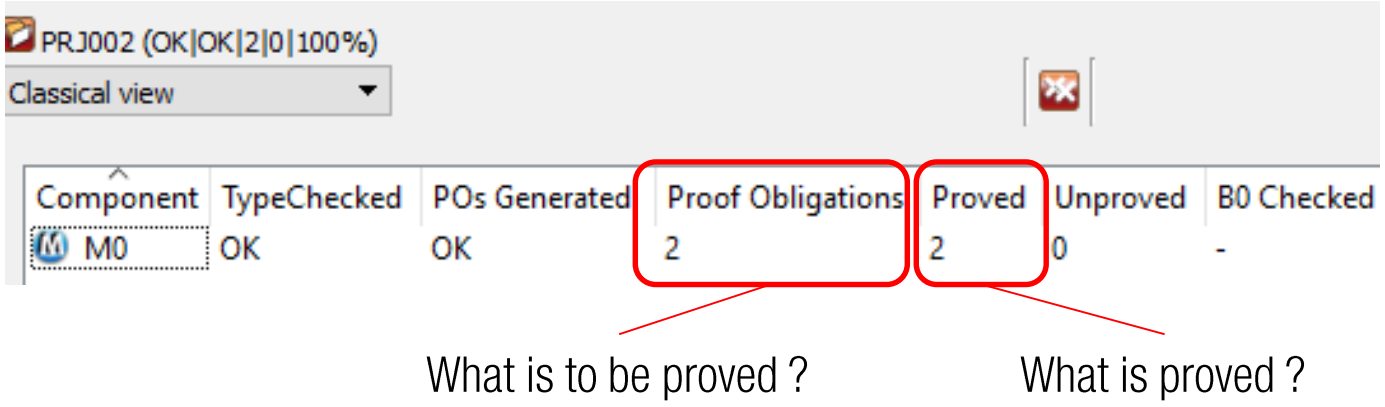
Operation

- 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
```

A Quick Example

Demo: project QuickExample

Summary

```
MACHINE
    M0
VARIABLES
    xx
INVARIANT
    xx: INT
INITIALISATION
    xx :: 1..5
OPERATIONS
    op =
    BEGIN
        xx :: 0..1
    END


END
```

```
IMPLEMENTATION M0_i
REFINES M0

CONCRETE_VARIABLES
    xx

INITIALISATION
    xx := 2

OPERATIONS
    op =
    BEGIN
        xx := 1
    END

END
```

```c
#include "M0.h"

/* Clause CONCRETE_CONSTANTS */
/* Basic constants */

/* Array and record constants */
/* Clause CONCRETE_VARIABLES */

static int32_t M0__xx;
/* Clause INITIALISATION */
void M0__INITIALISATION(void)
{

    M0__xx = 2;
}

/* Clause OPERATIONS */

void M0__op(void)
{
    M0__xx = 1;
}
```

QuickExample (OK|OK|4|0|100%)

Classical view

Everything proved

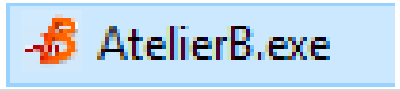
Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
M0	OK	OK	2	2	0	OK
M0_i	OK	OK	2	2	0	OK

Able to be translated

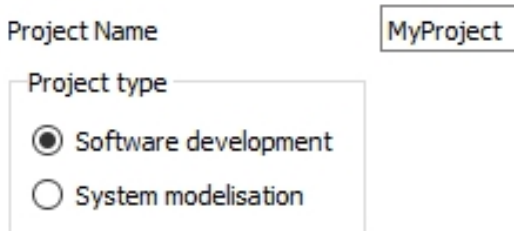
Something to prove

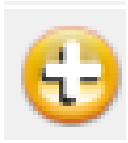
Your turn

- Start Atelier B

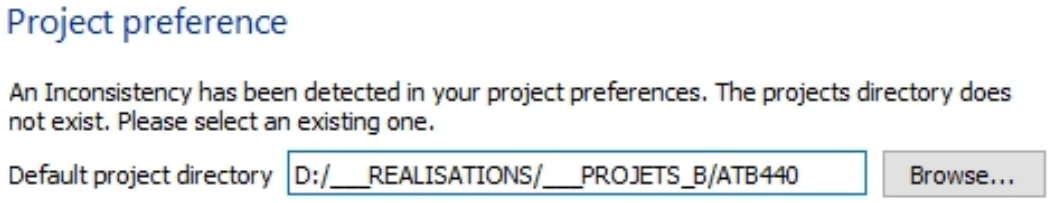


- Create a project

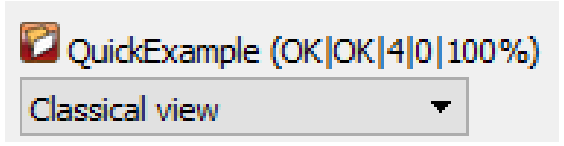




- Select a directory with write access



- Open the project (double click)
- Select « classical view »



- Create a component, a MACHINE named M0

- Select M0 then create a component, an IMPLEMENTATION of M0

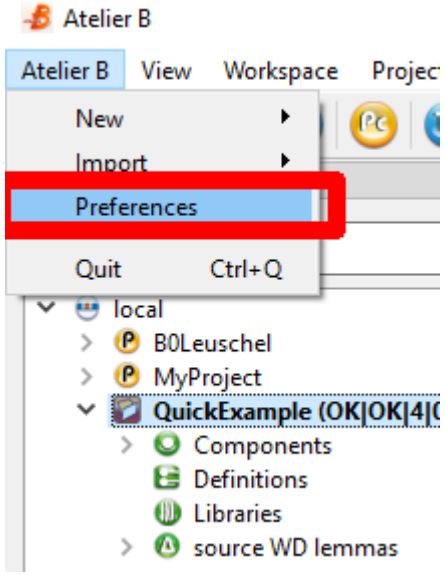
```
MACHINE
    M0
VARIABLES
    xx
INVARIANT
    xx: INT
INITIALISATION
    xx :: 1..5
OPERATIONS
    op =
    BEGIN
        xx :: 0..1
    END


END
```

```
IMPLEMENTATION M0_i
REFINES M0

CONCRETE_VARIABLES
    xx

INITIALISATION
    xx := 2

OPERATIONS
    op =
    BEGIN
        xx := 1
    END

END
```

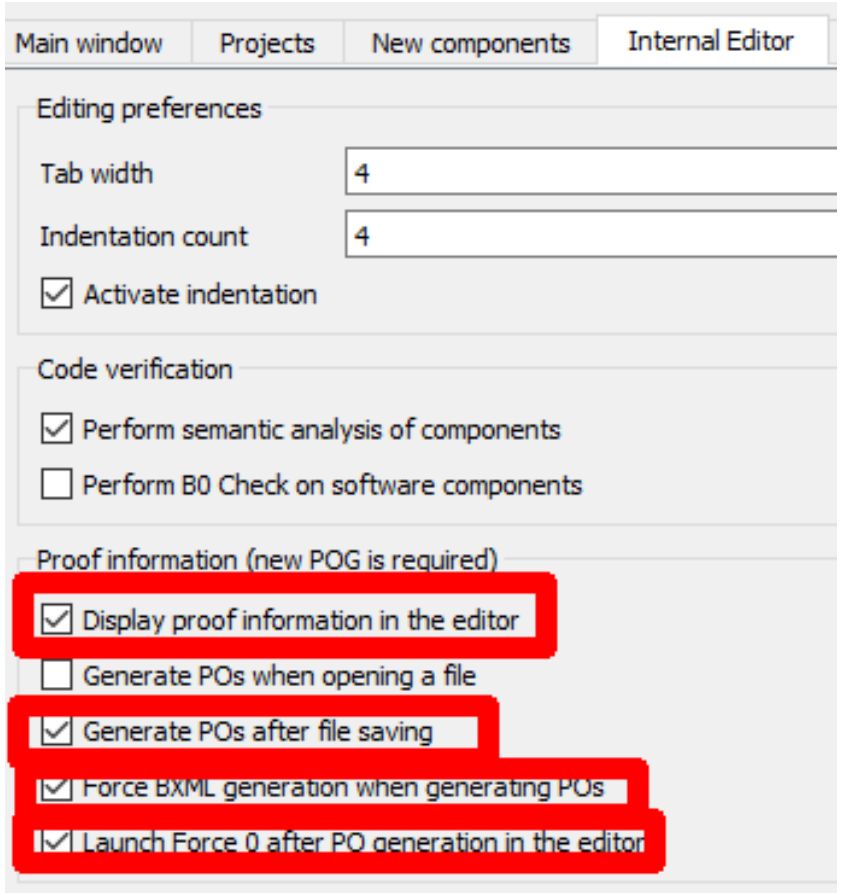
- Save (Ctrl+S) then Click on « F0 » (proof with force 0 prover)

How to get the colored editor ?

- Select Atelier B / Preferences menu item



- Select Internal Editor
- Check in the Proof obligation section as follows



- Close the open text editor
- Reopen it
- Ctrl+S to save
- Your editor should look like this:

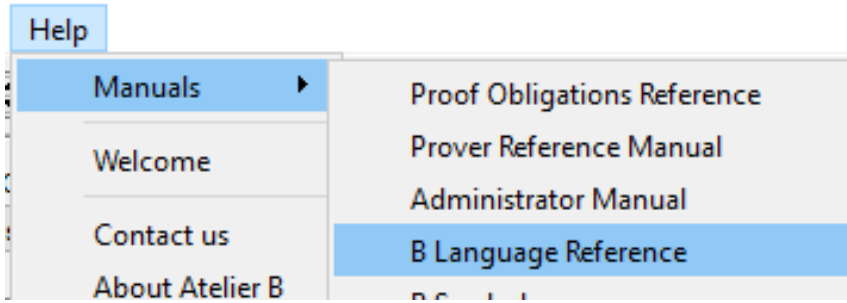
More Complex Examples

- Specification
- Refinement
- Implementation
- Architecture
- Proof
- Code generation

Foreword

- Only small part of the B language is being explored through this tutorial

- You are invited to have a look at the B language Reference manual

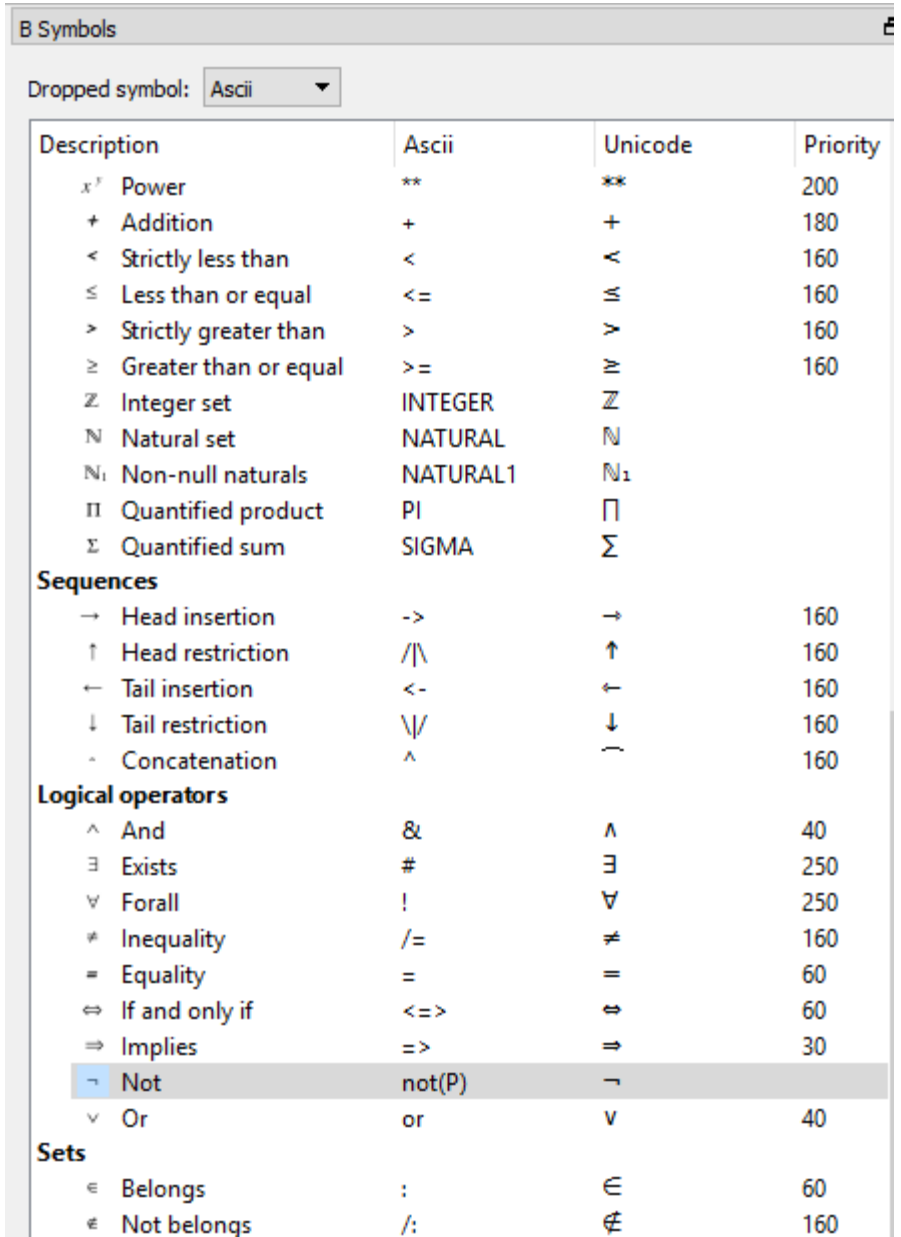


- Automatic proof is not fully automatic:
  - When it is proven, the model part is correct
  - When it is not proven, we don't know if the model part is correct
    - Either the model part is incorrect or the proof tool is not able to complete
    - Visual inspection is required

- We will address models that are mostly automatically proven

Foreword

- Models are text-based
- Mathematical notation with ASCII
- Either type in the ASCII
- Or drag & drop from the “B Symbols” view inside the B model editor



Key concepts

B models are models of software, by using

- **Predicates**
  - a predicate is a logical formula, which may or may not hold (is true or is false)
  - equations, inequalities and membership of a set are simple predicates

    e.g. $xx = 3$

    $5 < 2$

    $xx \in \{1, 2, 3\}$
  - simple predicates may be combined by negation, conjunction or disjunction

    e.g. $xx + yy = 0 \ \wedge \ xx < yy$

- **Substitutions**
  - substitutions represent the transformation of data by programs
  - so they change the state of a system
  - they concern some list of variables
  - substitutions are mathematically defined as predicate transformers

  e.g. $xx := 0$

  $xx :\in \text{INTEGER}$

  $xx, yy \in (xx \in \text{INTEGER} \wedge yy \in \text{INTEGER})$

Key concepts

B module is made of specification component

```
1-  MACHINE
2       M0
3-  ABSTRACT_CONSTANTS
4       C0
5-  PROPERTIES
6       C0 : INTEGER
7-  ABSTRACT_VARIABLES
8       xx
9-  INVARIANT
10      xx: BOOL
11- INITIALISATION
12      xx := FALSE
13- OPERATIONS
14      op1 =
15-     BEGIN
16          xx :: BOOL
17      END
18  END
```

Static

Dynamic

Implementation component

```
1   IMPLEMENTATION M0_i
2-  REFINES M0
3
4-  CONCRETE_VARIABLES
5       xx
6-  INITIALISATION
7       xx := FALSE
8-  OPERATIONS
9       op1 =
10-     BEGIN
11          xx := TRUE
12      END
13  END
```

Dynamic

Specification

```
1-  MACHINE
2       M0
3-  ABSTRACT_CONSTANTS
4       C0
5-  PROPERTIES
6       C0 : INTEGER
7-  ABSTRACT_VARIABLES
8       xx
9-  INVARIANT
10      xx: BOOL
11- INITIALISATION
12      xx := FALSE
13- OPERATIONS
14      op1 =
15-     BEGIN
16          xx :: BOOL
17      END
18  END
```

List of identifiers — Ex: c0, xx, vv, alt_ — 2 characters minimum

Predicates — Ex: c0 : BOOL & xx : INTEGER & …

List of identifiers

Predicates

Substitutions — Ex: c0 := TRUE || xx := 2

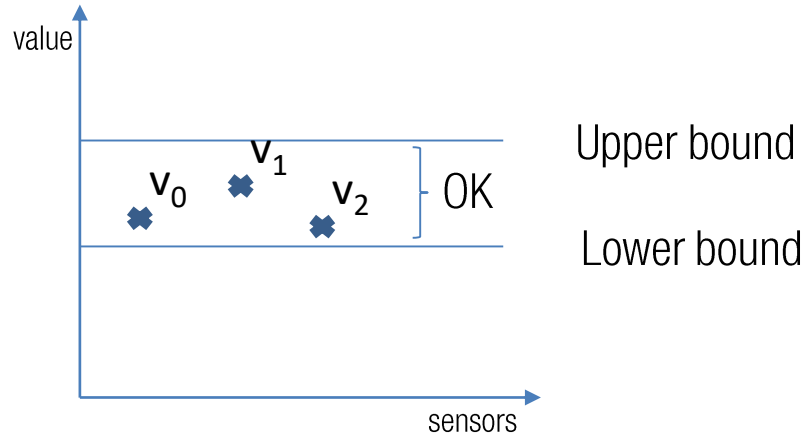
Substitutions

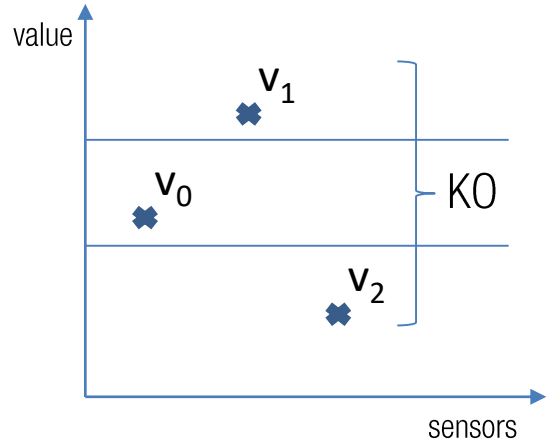
Specification: example 1

- 3 sensors return a measurement (whatever it is)
- if all the measurents are inside a range (consensus), then return OK else return KO

- Objective: specify the service

- We need:
  - 3 variables representing the measurements
  - The type and domain of the variables
  - Two constants defining the bounds for the range
  - A return value





Key concepts

- **Predicates** (some)

propositions

$\neg P$	negation of P (logical NOT)
$P \wedge Q$	conjunction of P and Q (logical AND)
$P \vee Q$	disjunction of P and Q (logical OR)
$P \Rightarrow Q$	logical implication: $\neg P \vee Q$
$P \Leftrightarrow Q$	logical equivalence: $P \Rightarrow Q \wedge Q \Rightarrow P$

quantified predicates

$\forall x . (P_x \Rightarrow Q_x)$	universal quantification
$\exists x . (P_x)$	existential quantification: $\neg (\forall x . (\neg P_x))$

Key concepts

- **Predicates** (some)

equality predicates

let x and y be two expressions

$x = y$	x equal to y
$x \neq y$	non equality: $\neg (x = y)$

inequality predicates

let x and y be two integer expressions

$x < y$	x strictly less than y
$x \leq y$	x less than or equal to y
$x > y$	x strictly greater than y
$x \geq y$	x greater than or equal to y

Key concepts

- **Predicates** (some)

set predicates

let x be an element, and let X and Y be two sets

$x \in X$	membership: x is an element of X
$x \notin X$	non membership

Specification: example 1

- 3 sensors return a measurement (whatever it is)
- if all the measurents are inside a range (consensus), then return OK else return KO

- Objective: specify the service

- We need:
  - 3 variables representing the measurements
  - The type and domain of the variables
  - Two constants defining the bounds for the range
  - A return value

```
1-  MACHINE
2       M0
3-  CONCRETE_CONSTANTS
4       LowerBound,
5       UpperBound
6-  PROPERTIES
7       LowerBound : NATURAL &
8       UpperBound : NATURAL &
9       LowerBound <= UpperBound
10  |
11  END
```

Actions:

- create a new project,
- create a component,
- type in the model above

Key concepts

- **Substitutions** (some)

null substitution [Spec, Impl]

skip the variables keep their values

“becomes equal to” substitution [Spec, Impl]

v := E the value of E is assigned to v

e.g. vv := 0

xx:= yy + 1

aa, bb := cc, dd

ff(ii) := mm

“becomes element of” substitution [Spec]

v :: X an element of X is assigned to v

e.g. vv :: (1..3)

Key concepts

- **Substitutions** (some)

simultaneous substitutions [Spec]

S1 || S2 applies the substitutions S1 and S2 simultaneously
the variables modified in S1 and S2 must be distinct

e.g. x := 1 || y := 2
x := y || y := x

Specification: example 1

- 3 sensors return a measurement (whatever it is)
- if all the measurents are inside a range (consensus), then return OK else return KO

- Objective: specify the service

- We need:
  - 3 variables representing the measurements
  - The type and domain of the variables
  - Two constants defining the bounds for the range
  - A return value

```
 1-      MACHINE
 2           M0
 3-      CONCRETE_CONSTANTS
 4           LowerBound,
 5           UpperBound
 6-      PROPERTIES
 7           LowerBound : NATURAL &
 8           UpperBound : NATURAL &
 9           LowerBound <= UpperBound
10-      ABSTRACT_VARIABLES
11           v0, v1, v2
12-      INVARIANT
13  1/1      v0 : NATURAL &
14           v1 : NATURAL &
15           v2 : NATURAL
16-      INITIALISATION
17  1/1      v0 := 0 ||
18           v1 := 0 ||
19           v2 := 0
20      |
21      END
```

Actions:

- type in the model above (complement)

Key concepts

- **Substitutions** (some)

"becomes such that" substitution [Spec]

x :(P_x) the variable x is assigned a value which satisfies the predicate P_x

e.g. $x :(x : NAT \ \& \ x \bmod 3 = 1)$

(results of dividing n by m, where $n : \mathbb{N}$ and $m : \mathbb{N}1$)

$q, r : (q : N \ \& \ r : N \ \& \ n = (m * q) + r \ \& \ r < m)$

the previous value of x can be referenced in P_x by x$0

e.g. $x :(x > x\$0)$

Key concepts

- **Substitutions** (some)

IF substitution [Spec, Impl]

IF P_1 THEN S_1 ELSIF P_2 THEN S_2 … ELSE S END

the substitution applied is:

- S_i if P_i holds **and** the previous predicates do not hold
- S if no predicate P_i holds (by default S is skip)

e.g.

```
IF  x > 10  THEN
    x := x - 10
ELSIF  x = 0  THEN
    x := x + 1
ELSE
    x := 1
END
```

Key concepts

- **Substitutions** (some)

BEGIN substitution (block substitution) [Spec, Impl]

BEGIN S END used to parenthesize substitutions

e.g. BEGIN $x := y \;||\; y := x$ END

BEGIN $x := y \;;\; y := x + 1$ END

Specification: example 1

- 3 sensors return a measurement (whatever it is)
- if all the measurents are inside a range (consensus), then return OK else return KO

- Objective: specify the service

- We need:
  - 3 variables representing the measurements
  - The type and domain of the variables
  - Two constants defining the bounds for the range
  - A return value

```
20-  OPERATIONS
21       update =
22-      BEGIN
23           v0 :: NATURAL ||
24           v1 :: NATURAL ||
25           v2 :: NATURAL
26       END
27       ;
28       return <-- check =
29-      BEGIN
30-          IF v0 >= LowerBound & v0 <= UpperBound &
31              v1 >= LowerBound & v1 <= UpperBound &
32              v2 >= LowerBound & v2 <= UpperBound
33-         THEN
34              return := TRUE
35-         ELSE
36              return := FALSE
37          END
38       END
39   END
```

Actions:

- type in the model above (complement)

Specification: example 1

```
20-  OPERATIONS
21       update =
22-      BEGIN
23           v0 :: NATURAL ||
24           v1 :: NATURAL ||
25           v2 :: NATURAL
26       END
27       ;
28       return <-- check =
29-      BEGIN
30-          IF v0 >= LowerBound & v0 <= UpperBound &
31              v1 >= LowerBound & v1 <= UpperBound &
32              v2 >= LowerBound & v2 <= UpperBound
33-         THEN
34              return := TRUE
35-         ELSE
36              return := FALSE
37          END
38       END
39   END
```

```
21       update =
22-      BEGIN
23           v0, v1, v2 : (v0: NATURAL & v1: NATURAL & v2: NATURAL)
24       END
```

Different ways of modelling

```
28       return <-- check =
29-      BEGIN
30-          return := bool(v0 >= LowerBound & v0 <= UpperBound &
31              v1 >= LowerBound & v1 <= UpperBound &
32              v2 >= LowerBound & v2 <= UpperBound )
33       END
```

```
26       return <-- check =
27-      BEGIN
28-          return := bool(
29              v0 : LowerBound..UpperBound &
30              v1 : LowerBound..UpperBound &
31              v2 : LowerBound..UpperBound )
32       END
```

Key concepts

- **Substitutions** (some)

sequential substitutions [Impl]

S1 ; S2 applies the substitution S1 and then the substitution S2

e.g. $x := 1 \; ; \; y := 2$

$x := y \; ; \; y := x + 1$

Specification: example 1

```
IMPLEMENTATION M0_i
REFINES M0

VALUES
2/2     LowerBound = 10;
2/2     UpperBound = 20
CONCRETE_VARIABLES
3/3     v0, v1, v2
INITIALISATION
        v0 := 0;
        v1 := 0;
        v2 := 0
OPERATIONS
        update =
        BEGIN
1/1         v0 := 0; v1 := 0; v2 := 0
        END
        ;
2/2     return <-- check =
        BEGIN
            IF v0 >= LowerBound & v0 <= UpperBound &
                v1 >= LowerBound & v1 <= UpperBound &
                v2 >= LowerBound & v2 <= UpperBound
            THEN
1/1             return := TRUE
            ELSE
1/1             return := FALSE
            END
        END

END
```

Constants are given values that should verify their properties

Implementation requires to have only concrete variables

Should refine initialisation from specification model

Minimum implementation, we will see later on how to obtain random numbers

Should refine operation check from specification model

Actions:

- select specification component
- click « Add component » and select implementation
- type in the model above (complement)
- check that everything is proved

Key concepts

B module is made of specification component

```
1-  MACHINE
2       M0
3-  ABSTRACT_CONSTANTS
4       C0
5-  PROPERTIES
6       C0 : INTEGER
7-  ABSTRACT_VARIABLES
8       xx
9-  INVARIANT
10      xx: BOOL
11- INITIALISATION
12      xx := FALSE
13- OPERATIONS
14      op1 =
15-     BEGIN
16          xx :: BOOL
17      END
18  END
```

Implementation component

```
1   IMPLEMENTATION M0_i
2-  REFINES M0
3
4-  CONCRETE_VARIABLES
5       xx
6-  INITIALISATION
7       xx := FALSE
8-  OPERATIONS
9       op1 =
10-     BEGIN
11          xx := TRUE
12      END
13  END
```

Operations signatures have to be strictly identical between specification and implementation

Key concepts

Abstract constants and variables to support formal reasoning

```
1-  MACHINE
2       M0
3-  ABSTRACT_CONSTANTS
4       C0
5-  PROPERTIES
6       C0 : INTEGER
7-  ABSTRACT_VARIABLES
8       xx
9-  INVARIANT
10      xx: BOOL
11- INITIALISATION
12      xx := FALSE
13- OPERATIONS
14      op1 =
15-     BEGIN
16          xx :: BOOL
17      END
18  END
```

Abstract constants and variables only visible in the component where they are defined

In this case
xx from M0 and xx from M0_i
Are supposed to refer to the same variable

Only concrete constants and variables in implementation

```
1   IMPLEMENTATION M0_i
2-  REFINES M0
3
4-  CONCRETE_VARIABLES
5       xx
6-  INITIALISATION
7       xx := FALSE
8-  OPERATIONS
9       op1 =
10-     BEGIN
11          xx := TRUE
12      END
13  END
```

Except if they are defined as concrete

Modelling style: copy-paste

```
1-  MACHINE
2       N0
3-  CONSTANTS
4       LowerBound,UpperBound
5-  PROPERTIES
6       LowerBound: NATURA
7       LowerBound <= Uppe
8-  VARIABLES
9       v0, v1, v2
10- INVARIANT
11 1/1  v0, v1,
12- INITIALISAT N
13 1/1  v0 :=
14 1/1  v1 :=
15 1/1  v2 :=
16- OPERATIONS
17      return <-- check =
18-     BEGIN
19-         IF v0 >= LowerBound & v0 <= UpperBound &
20              v1 >= LowerBound & v1 <= UpperBound &
21              v2 >= LowerBound & v2 <= UpperBound
22-         THEN
23              return := TRUE
24-         ELSE
25              return := FALSE
26          END
27      END
28  END
```

OPERATIONS bodies are identical:
What is proved is ….. the copy-paste

```
1       IMPLEMENTATION N0_i
9           v0 := 0;
10          v1 := 0;
11          v2 := 0
12-     OPERATIONS
13 2/2      return <-- check =
14-         BEGIN
15-             IF v0 >= LowerBound & v0 <= UpperBound &
16                  v1 >= LowerBound & v1 <= UpperBound &
17                  v2 >= LowerBound & v2 <= UpperBound
18-             THEN
19 1/1              return := TRUE
20-             ELSE
21 1/1              return := FALSE
22              END
23          END
24
25      END
```

Modelling style: copy-paste

Actions:

- Replace check operation specification by the one below
- Verify that everything is still proved

```
26      return <-- check =
27-     BEGIN
28-         return := bool(
29              v0 >= LowerBound & v0 <= UpperBound &
30              v1 >= LowerBound & v1 <= UpperBound &
31              v2 >= LowerBound & v2 <= UpperBound )
32      END
```

```
 1      IMPLEMENTATION N0_i
 2      REFINES N0
 3-     VALUES
 4  2/2     LowerBound = 10;
 5  2/2     UpperBound = 20
 6-     CONCRETE_VARIABLES
 7          v0, v1, v2
 8-     INITIALISATION
 9          v0 := 0;
10          v1 := 0;
11          v2 := 0
12-     OPERATIONS
13  2/2     return <-- check =
14-         BEGIN
15-             IF v0 >= LowerBound & v0 <= UpperBound &
16                  v1 >= LowerBound & v1 <= UpperBound &
17                  v2 >= LowerBound & v2 <= UpperBound
18-             THEN
19  1/1             return := TRUE
20-             ELSE
21  1/1             return := FALSE
22              END
23          END
24
25      END
```

Modelling style: empty specification

The specification is empty:

- No variable, no invariant
- Operation is skip (model variables unchanged)

```
1-  MACHINE
2       N0
3-  OPERATIONS
4       op1 = skip
5   END
```

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved
M0	OK	OK	0	0	0
M0_i	OK	OK	1	1	0
N0	OK	OK	0	0	0
N0_i	OK	OK	0	0	0

1 proof obligation

0 proof obligation

Implementation component

```
1   IMPLEMENTATION N0_i
2-  REFINES N0
3
4-  CONCRETE_VARIABLES
5       xx
6-  INVARIANT
7       xx: BOOL
8-  INITIALISATION
9       xx := FALSE
10- OPERATIONS
11      op1 =
12-     BEGIN
13          xx := TRUE
14      END
15  END
```

Software with B: importing components

- Restrictions
  - An operation can't call another operation of the same module

```
1  MACHINE
2      M0
3  OPERATIONS
4      op1 = skip;
5      op2 = skip;
6      op3 = skip
7  END
```

```
1  IMPLEMENTATION M0_i
2  REFINES M0
3
4  OPERATIONS
5      op1 =
6      BEGIN
7          op2;
8          op3
9      END
10 END
```

  - Hence operations have to call other operations from imported components

```
1  MACHINE
2      M0
3  OPERATIONS
4      op1 = skip
5  END
```

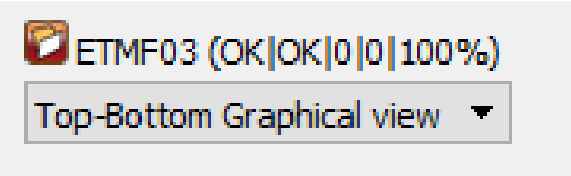
```
1  IMPLEMENTATION M0_i
2  REFINES M0
3  IMPORTS N0
4  OPERATIONS
5      op1 =
6      BEGIN
7          op2;
8          op3
9      END
10 END
```

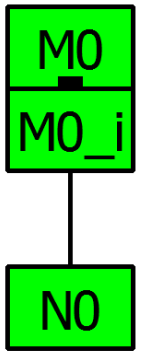
```
1  MACHINE
2      N0
3  OPERATIONS
4      op2 = skip;
5      op3 = skip
6  END
```

Software with B : importing components

- Imported services can be used from IMPLEMENTATION. Only their specification is visible, not the way they are implemented

- Project dependency graph is as follows (graphical view)



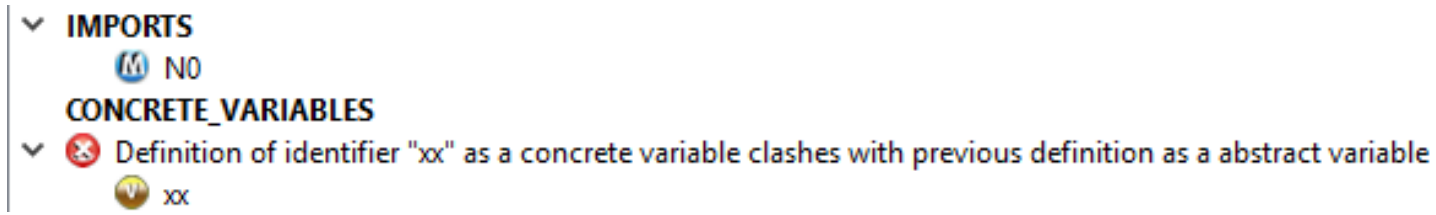


Software with B : importing components

- Restrictions
  - Concrete variables are initialized in one and only one implementation

```
1   IMPLEMENTATION M0_i
2   REFINES M0
3   IMPORTS N0
4-  CONCRETE_VARIABLES
5       xx
6-  INVARIANT
7       xx: BOOL
8-  INITIALISATION
9       xx := TRUE
10- OPERATIONS
11      op1 =
12-     BEGIN
13          op2;
14          op3
15      END
16  END
```

```
1-  MACHINE
2       N0
3-  ABSTRACT_VARIABLES
4       xx
5-  INVARIANT
6       xx: BOOL
7-  INITIALISATION
8       xx := TRUE
9-  OPERATIONS
10      op2 = skip;
11      op3 = skip
12  END
```



Software with B : context machines

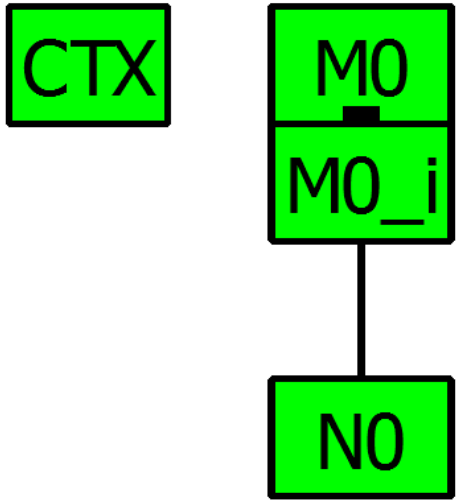
- Advice for clean structure
  - Constants and sets definitions have to be regrouped in context machines that can be seen (and used by different other machines)

```
1-  MACHINE
2       CTX
3-  CONSTANTS
4       LOWERBOUND,
5       UPPERBOUND
6-  PROPERTIES
7       LOWERBOUND : NAT &
8       UPPERBOUND : NAT
9   END
10
```

```
1-  MACHINE
2       M0
3   SEES CTX
4-  OPERATIONS
5       op1 = skip
6   END
7
```

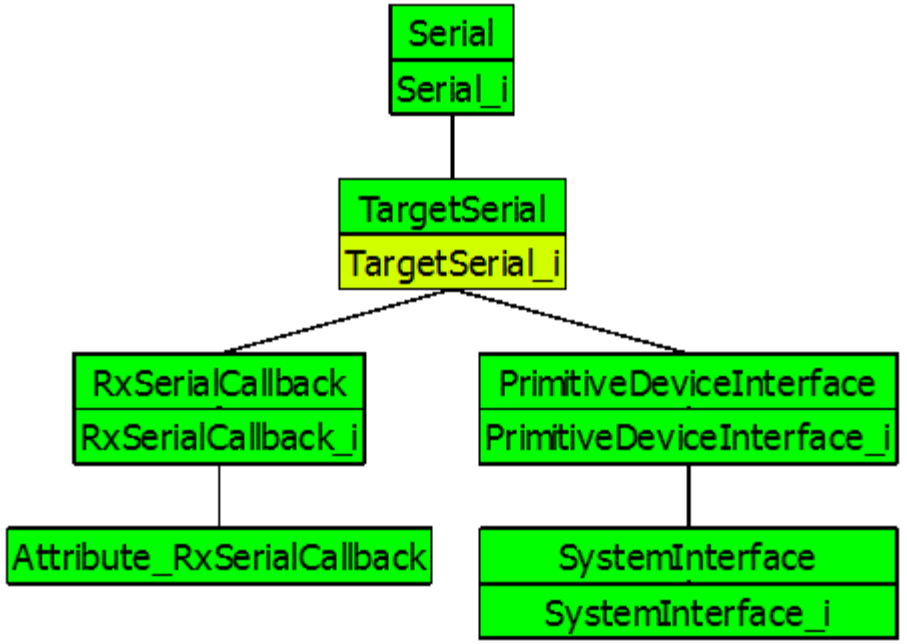
```
1-  MACHINE
2       N0
3   SEES CTX
4-  OPERATIONS
5       op2 = skip;
6       op3 = skip
7   END
```

```
1   IMPLEMENTATION M0_i
2   REFINES M0
3   IMPORTS N0
4   SEES CTX
5-  CONCRETE_VARIABLES
6       xx
7-  INVARIANT
8       xx: BOOL
9-  INITIALISATION
10      xx := TRUE
11- OPERATIONS
12      op1 =
13-     BEGIN
14          op2;
15          op3
16      END
17  END
```



Software with B

- 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)

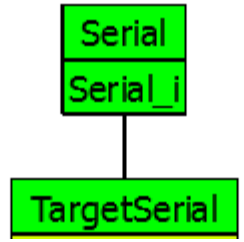


Tree-like structure of a B project
(Import links)

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

Software with B

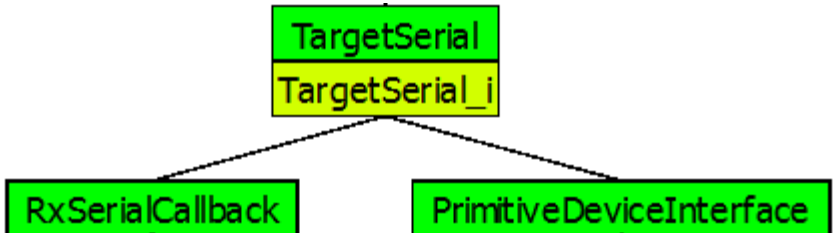
Proving Serial_i



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

Software with B

Proving TargetSerial_i



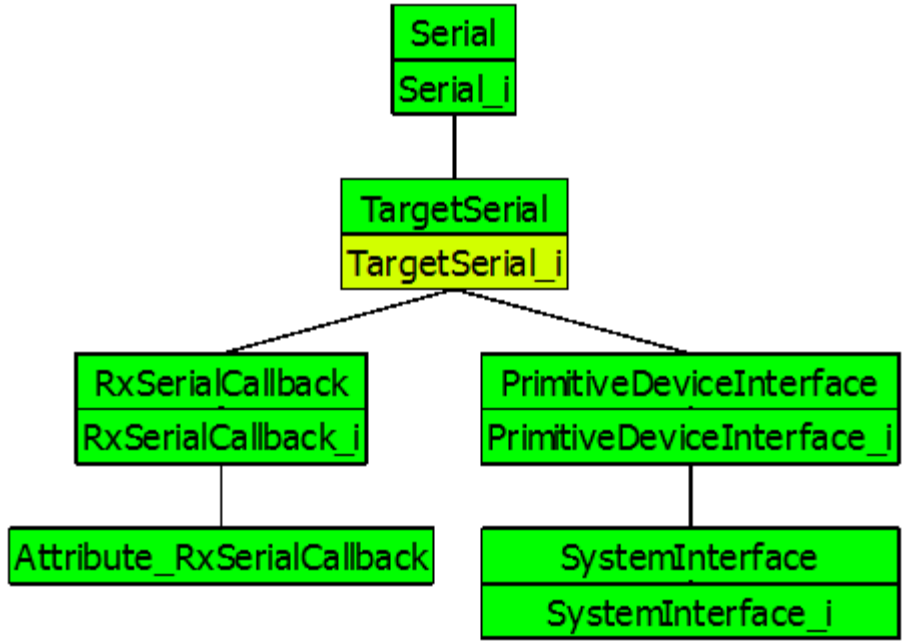
- Proving an **IMPLEMENTATION** requires:
  - Its specification **MACHINE**
  - The specification of the **MACHINE**s it **IMPORTS**
  - The specification of the **MACHINE**s it **SEES**

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.

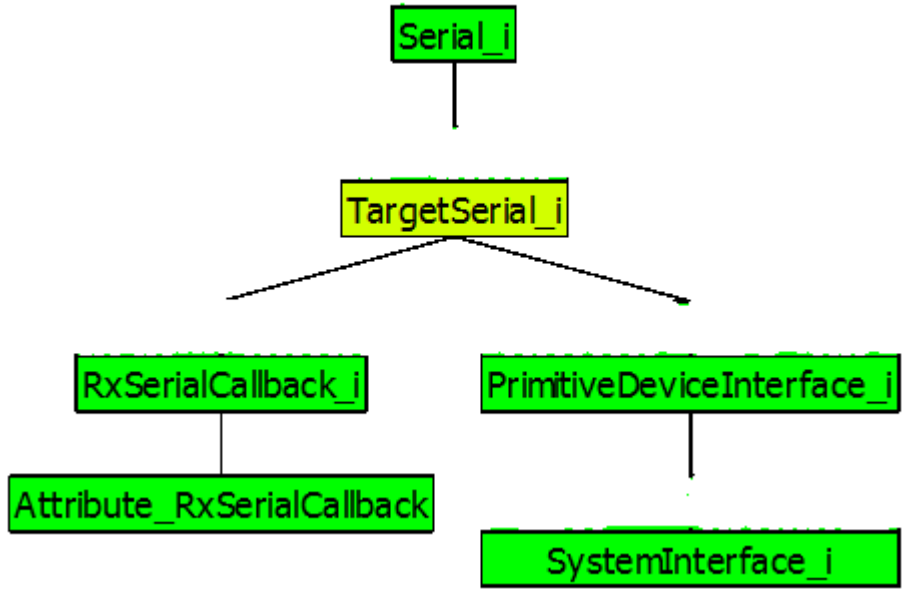
Translated code is very close to original B implementation



Software with B

In a B project, only implementations are translated

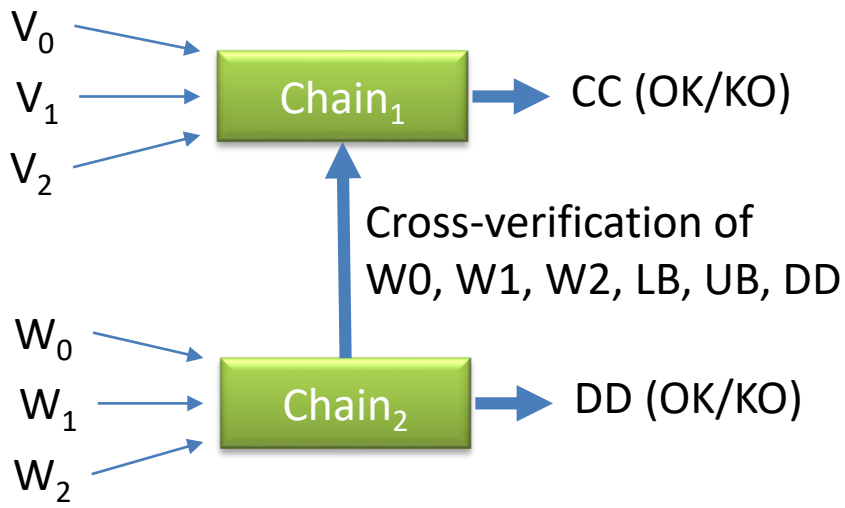
- Implementation have to be implementable
- No more non-determinism, sets, functions, etc.
- Only implementable types (BOOL, INT, tables of BOOL and INT)
- Only imperative constructs: valuation, IF THEN ELSE, CASE, WHILE, sequence, operation call.



Translated code is very close to original B implementation

Specification: example 2

- 3 sensors return a measurement (whatever it is)
- if all the measurents are inside a range (consensus), and if decision made by other chain is correct and identical, then return OK else return KO

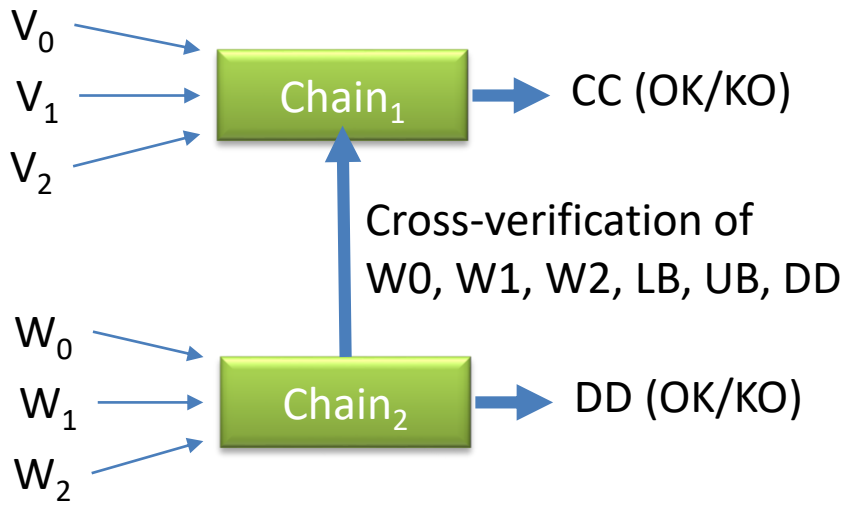


- Objective: specify the service

- Remarks:
  - W0, W1, W2 come from different sensors than V0, V1, V2
  - $Chain_2$ may be parameterized with different lower and upper bounds
  - $Chain_2$ may be programmed with different evaluation function
  - We are testing both data and program integrity (we could add counters incremented at each verification to demonstrate that the chains are still alive)

Specification: example 2

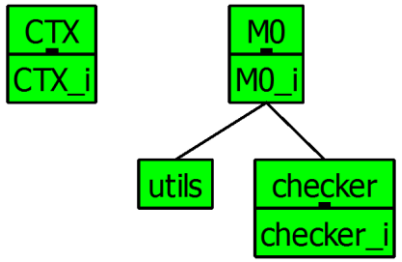
```
1-      MACHINE
2           M0
3-      VARIABLES
4           sane,
5           sane1, sane2
6-      INVARIANT
7           sane : BOOL &
8           sane1 : BOOL &
9           sane2: BOOL &
10  1/1     sane = bool(sane1=TRUE & sane2=TRUE)
11-     INITIALISATION
12  1/1     sane := FALSE ||
13  1/1     sane1 := FALSE ||
14  1/1     sane2 := FALSE
15-     OPERATIONS
16          check2 =
17-         BEGIN
18-             sane, sane1, sane2: (
19                  sane : BOOL &
20                  sane1 : BOOL &
21                  sane2: BOOL &
22                  sane = bool(sane1=TRUE & sane2=TRUE)
23              )
24          END
25
26      END
```



Actions:

- Create a new project
- Add component M0

```
1    IMPLEMENTATION M0_i
2    REFINES M0
3    SEES CTX
4-   IMPORTS utils, checker
5
6-   CONCRETE_VARIABLES
7  2/2     sane,
8  2/2     sane1, sane2
9
10-  INITIALISATION
11       sane := FALSE;
12       sane1 := FALSE;
13       sane2 := FALSE
14
15-  OPERATIONS
16       check2 =
17-2/2   VAR v0, v1, v2, w0, w1, w2, LB2, UB2, DD IN
18 2/2        v0, v1, v2 <-- getValues;
19 2/2        w0, w1, w2, LB2, UB2, DD <-- getOtherValues;
20 2/2        sane1 <-- check(v0, v1, v2);
21 2/2        sane2 <-- complete_check(w0, w1, w2, LB2, UB2, DD);
22            IF sane1 = TRUE & sane2 = TRUE
23-           THEN
24 1/1             sane := TRUE
25-           ELSE
26 1/1             sane := FALSE
27            END
28       END
29
30   END
```



Actions:

- Add component M0_i
- Add component CTX (contains the definition of the lower and upper bound constants)
- Add the component checker that defines check and complete_check operations
- Add the component utils that defines getValues and getOtherValues
- Add CTX_i component that gives VALUES to the constants

```
MACHINE
    utils
OPERATIONS
    v0, v1, v2 <-- getValues =
    BEGIN


    END
```

```
MACHINE
    checker
SEES CTX

OPERATIONS
    sane <-- check(v0, v1, v2) =
    PRE
        v0: NAT &
        v1: NAT &
        v2: NAT
    THEN


    END
```

Specification: example 2

```
MACHINE
    CTX
CONSTANTS
    LOWER_BOUND,
    UPPER_BOUND
PROPERTIES
    LOWER_BOUND : NAT &
    UPPER_BOUND : NAT &
    LOWER_BOUND <= UPPER_BOUND
END
```

```
1    IMPLEMENTATION CTX_i
2    REFINES CTX
3-   VALUES
4    2/2     LOWER_BOUND = 10;
5    2/2     UPPER_BOUND = 20
6
7     END
```

```
 1- MACHINE
 2      utils
 3- OPERATIONS
 4      v0, v1, v2 <-- getValues =
 5-     BEGIN
 6          v0 :: NAT ||
 7          v1 :: NAT ||
 8          v2 :: NAT
 9      END
10      ;
11      w0, w1, w2, LB, UB, DD <-- getOtherValues =
12-     BEGIN
13-         w0, w1, w2, LB, UB, DD: (
14              w0 : NAT &
15              w1 : NAT &
16              w2 : NAT &
17              LB : NAT &
18              UB : NAT &
19              DD : BOOL
20          )
21
22      END
23 END
```

```
MACHINE
    checker
SEES CTX

OPERATIONS
    sane <-- check(v0, v1, v2) =
    PRE
        v0: NAT &
        v1: NAT &
        v2: NAT
    THEN
        sane := bool(v0 >= LOWER_BOUND & v0 <= UPPER_BOUND &
            v1 >= LOWER_BOUND & v1 <= UPPER_BOUND &
            v2 >= LOWER_BOUND & v2 <= UPPER_BOUND)
    END
    ;
    sane <-- complete_check(w0, w1, w2, LB, UB, DD) =
    PRE
        w0: NAT &
        w1: NAT &
        w2: NAT &
        LB: NAT &
        UB: NAT &
        DD : BOOL
    THEN
        sane := bool(DD = bool(w0 >= LB & w0 <= UB &
            w1 >= LB & w1 <= UB &
            w2 >= LB & w2 <= UB))
        END
END
```

```
IMPLEMENTATION checker_i
REFINES checker

SEES CTX


OPERATIONS
    sane <-- check(v0, v1, v2) =
    BEGIN
        sane := bool(v0 >= LOWER_BOUND & v0 <= UPPER_BOUND &
            v1 >= LOWER_BOUND & v1 <= UPPER_BOUND &
            v2 >= LOWER_BOUND & v2 <= UPPER_BOUND)
    END
    ;

    sane <-- complete_check(w0, w1, w2, LB, UB, DD) =
    BEGIN
        sane := bool(DD = bool(w0 >= LB & w0 <= UB &
                w1 >= LB & w1 <= UB &
                w2 >= LB & w2 <= UB))
    END

END
```

How to complete the development

```
MACHINE
    utils
OPERATIONS
    v0, v1, v2 <-- getValues =
    BEGIN
        v0 :: NAT ||
        v1 :: NAT ||
        v2 :: NAT
    END
    ;
    w0, w1, w2, LB, UB, DD <-- getOtherValues =
    BEGIN
        w0, w1, w2, LB, UB, DD: (
            w0 : NAT &
            w1 : NAT &
            w2 : NAT &
            LB : NAT &
            UB : NAT &
            DD : BOOL
        )

    END
END
```

Actions:

- Generate code for utils component (utils.h) is generated
- Copy-paste and rename the file in utils.c
- Edit such as:

```c
#include "M0.h"

void utils__INITIALISATION(void) {};

/* Clause OPERATIONS */

void utils__getValues(int32_t *v0, int32_t *v1, int32_t *v2) {
    *v0 = 12;
    *v1 = 14;
    *v2 = 18;
};

void utils__getOtherValues(int32_t *w0, int32_t *w1, int32_t *w2, int32_t *LB, int32_t *UB, bool *DD) {
    *w0 = 13;
    *w1 = 19;
    *w2 = 12;
    *LB = 10;
    *UB = 20;
    *DD = true;

};
```

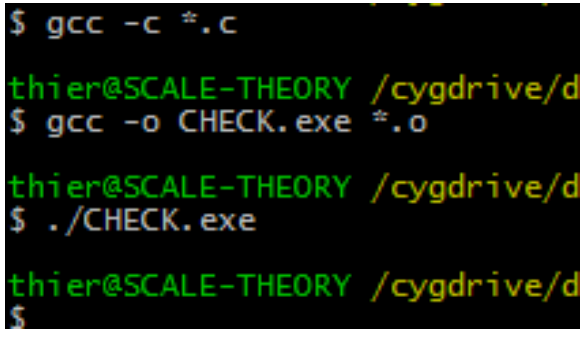
How to complete the development

Actions:

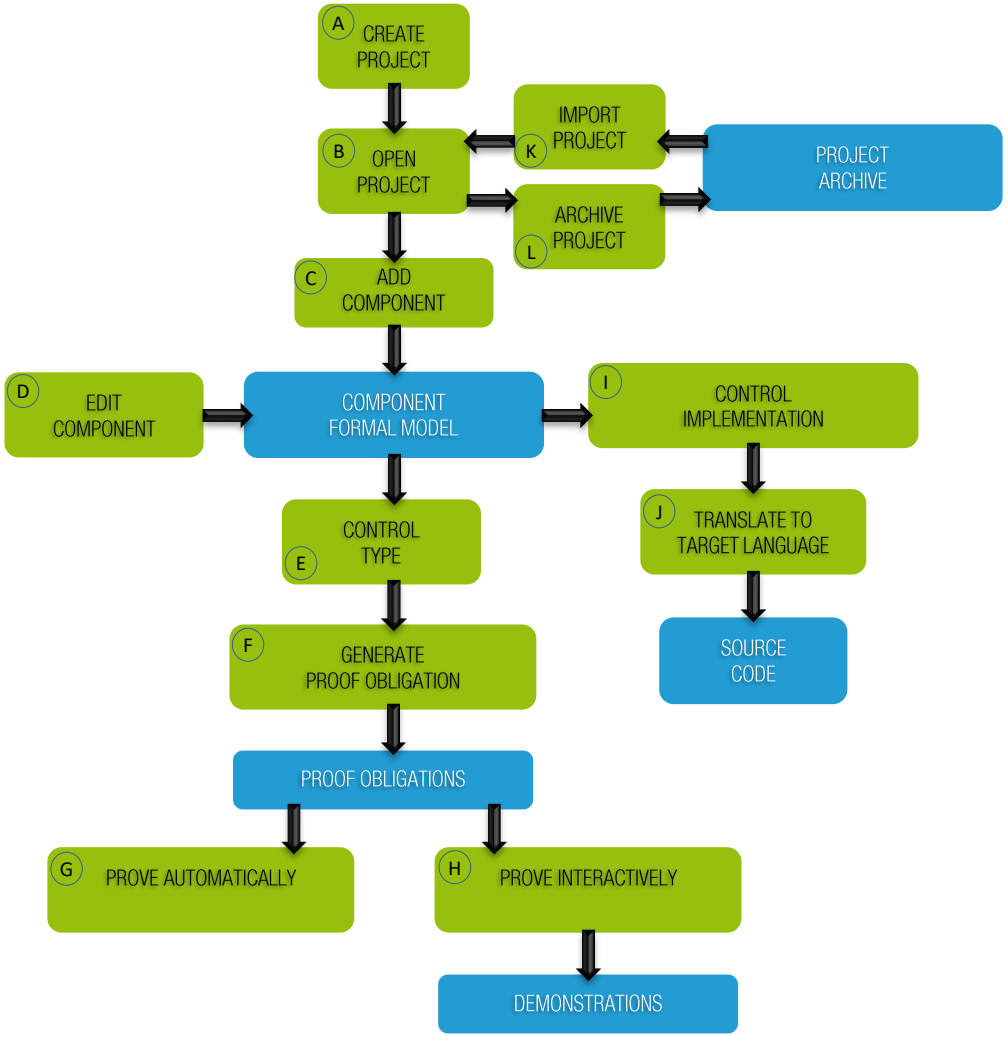
- Create a top.c file such as:

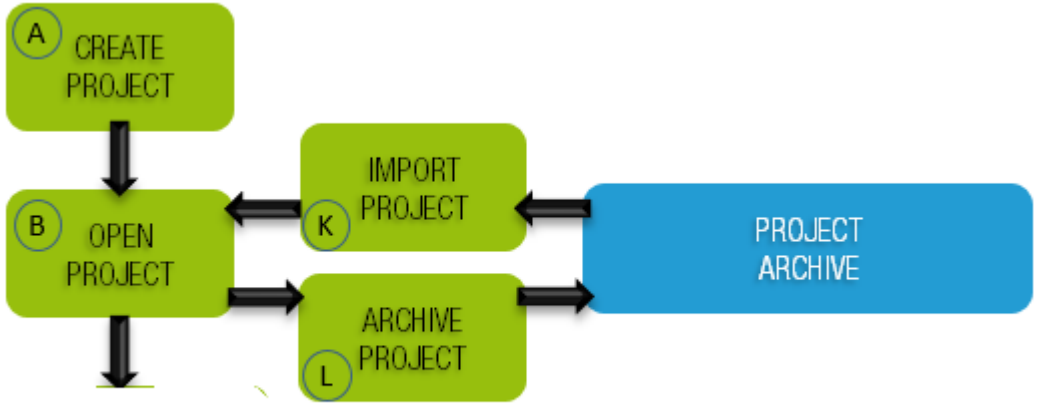
```
1
2 #include "M0.h"
3 void main (void){
4     M0__INITIALISATION();
5     M0__check2();
6 }
```

- Compile the file
  - Gcc –c *.c
  - Gcc –o CHECK.exe *.o

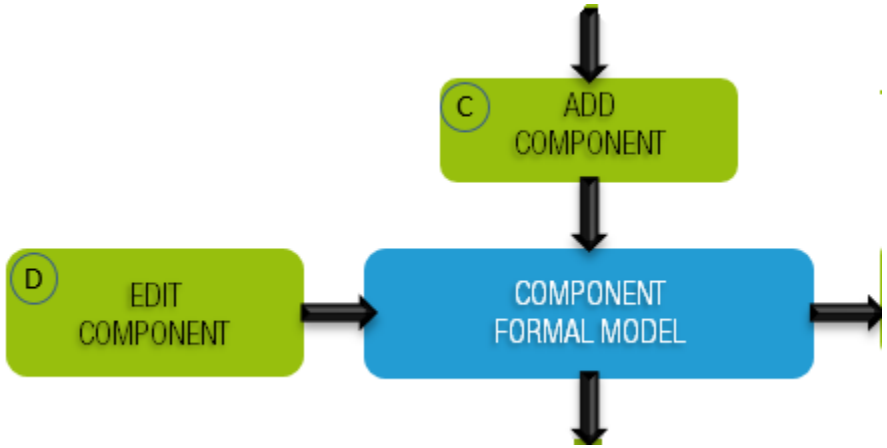


Development processus



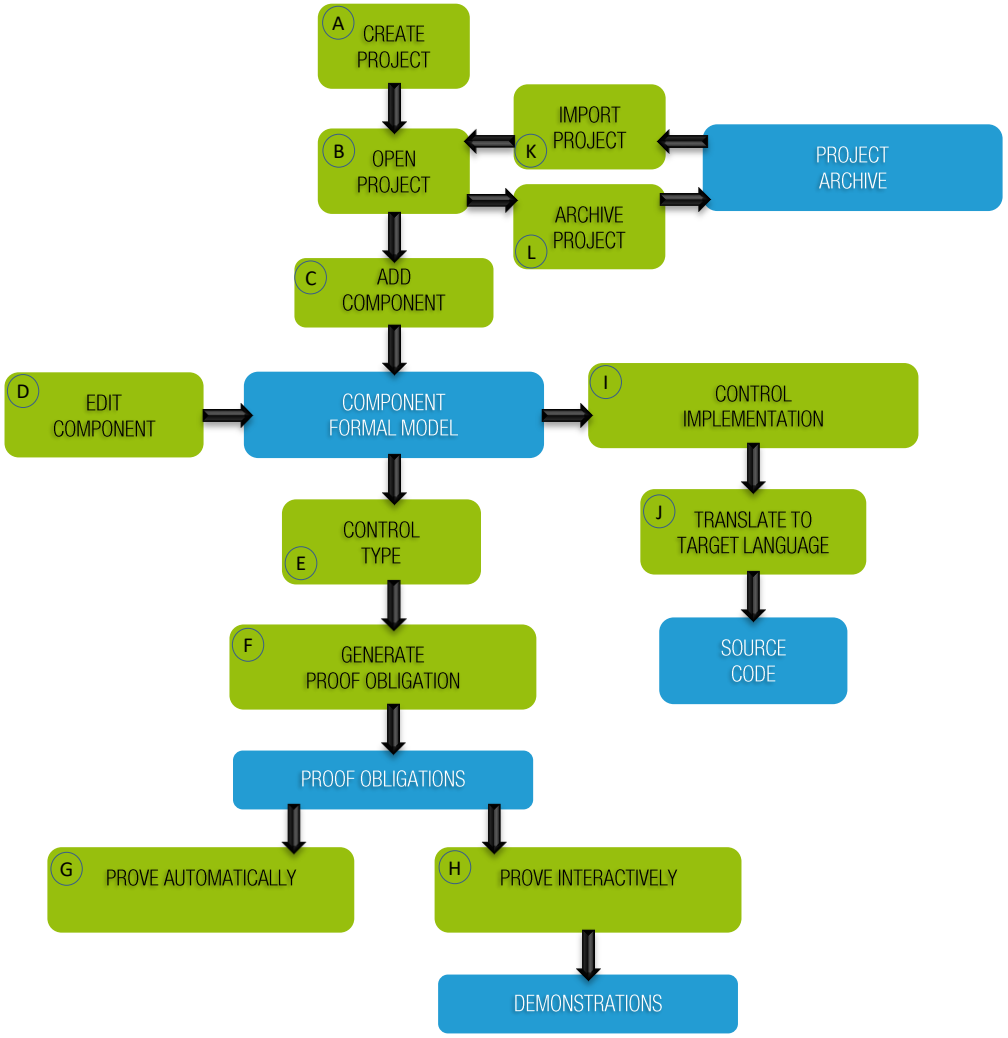


Project management



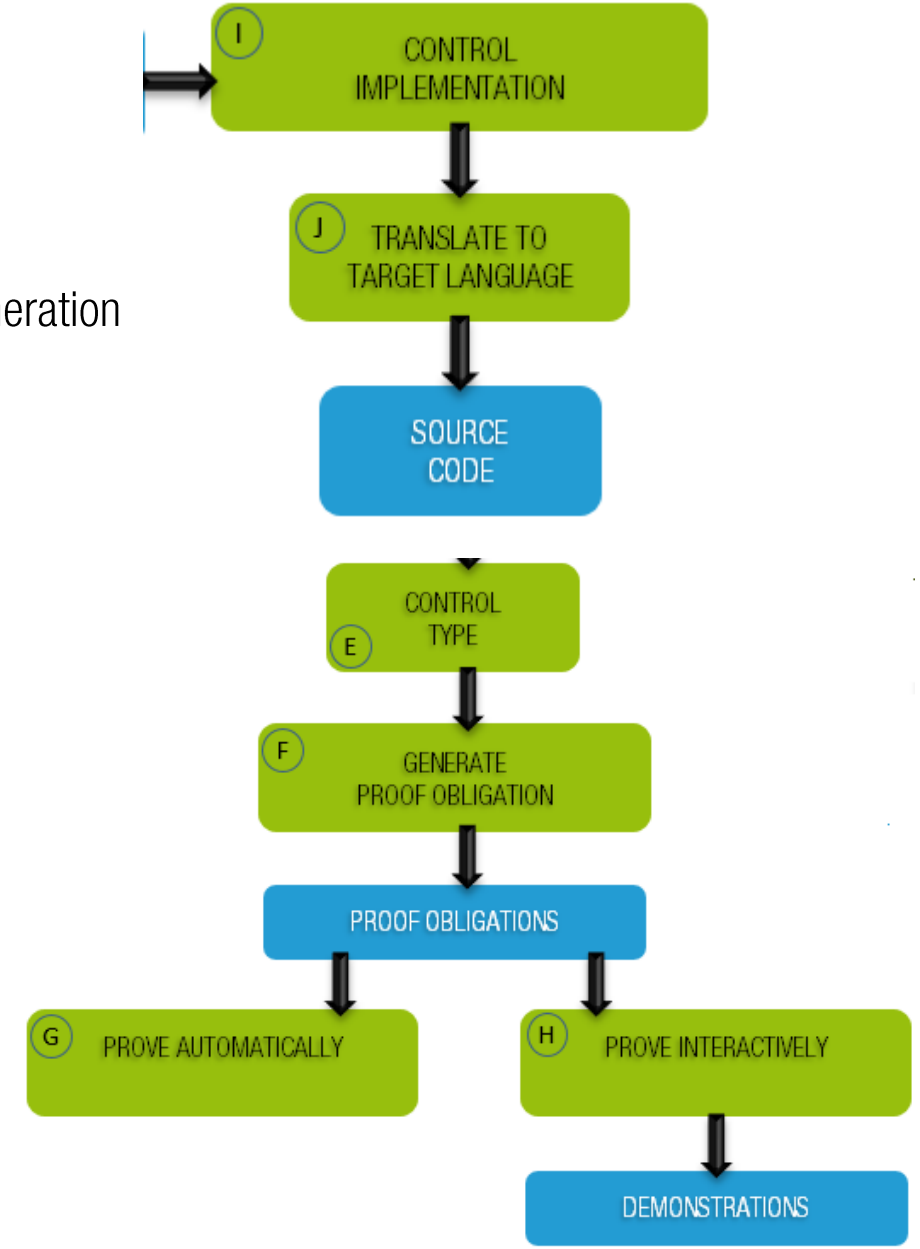
Modelling

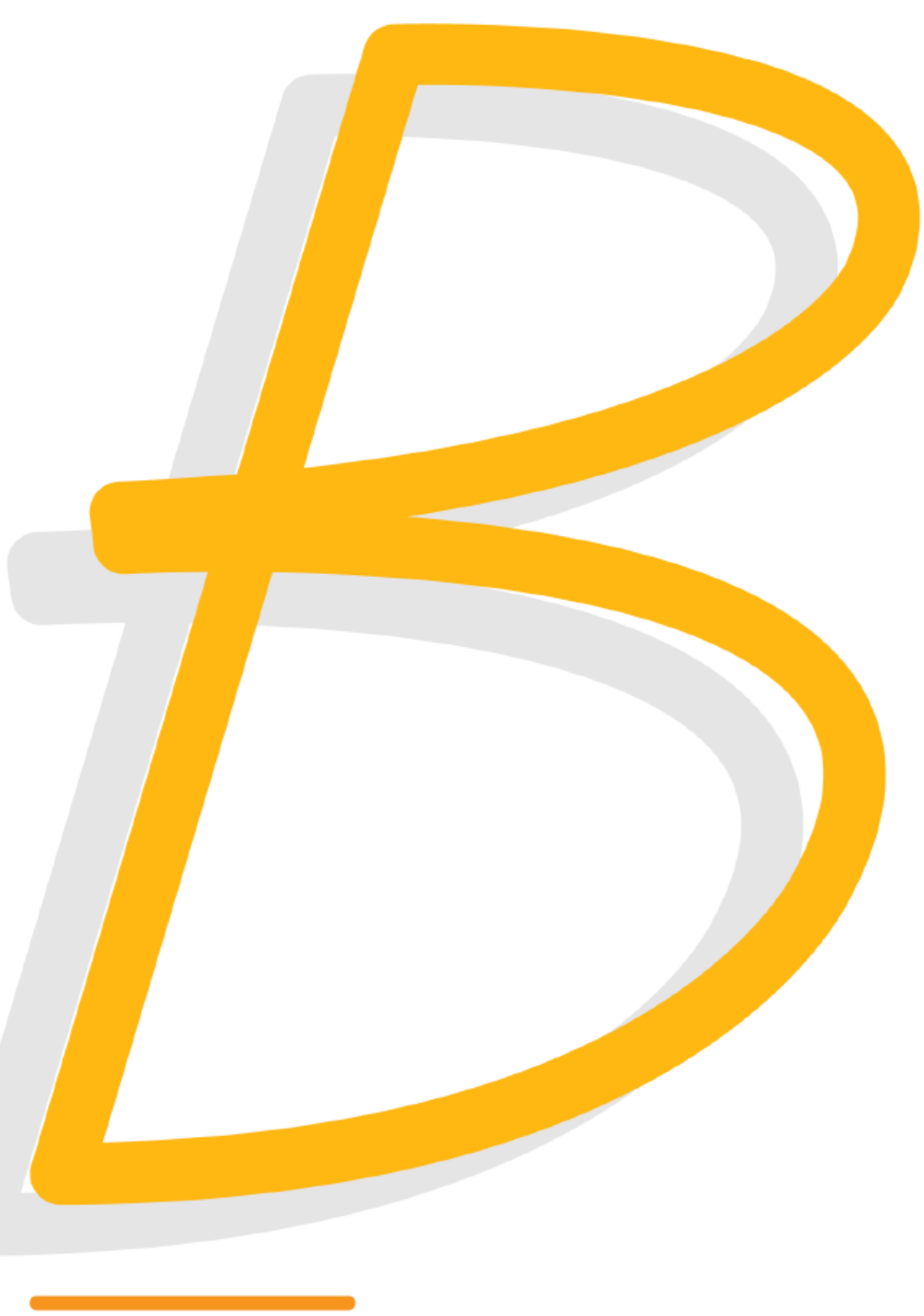
Development process



Code generation

Proving





Thank you for your attention

Thierry Lecomte
ETMF 2016
Natal

CLEARSY
SYSTEM ENGINEERING