# CLEARSY Safety Platform For Education

*Presentation*
*Hands-on*

**Thierry Lecomte** *(thierry.lecomte@clearsy.com)*

# Agenda

**Introduction to the CLEARSY Safety Platform**

**Development process (demo video)**
**Bits of B**
**Using the modelling interface**

**The clock example (synchronous)**
**The combinatorial example (asynchronous)**

**Conclusion**

# Agenda

**Introduction to the CLEARSY Safety Platform**

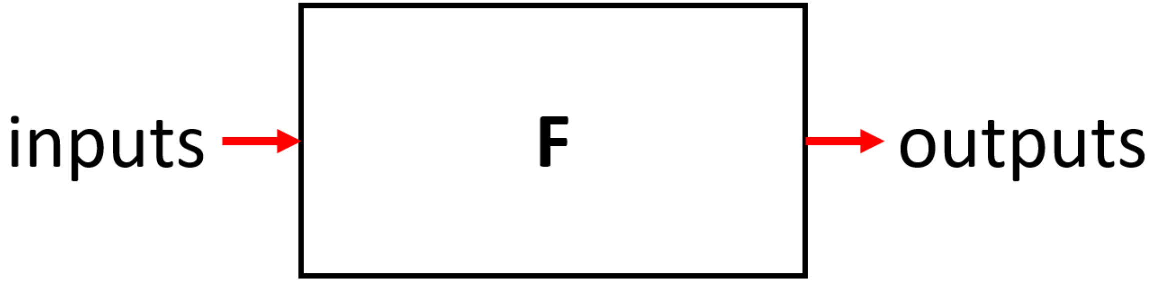
Development process (demo video)
Bits of B
Using the modelling interface

The clock example (synchronous)
The combinatorial example (asynchronous)

Conclusion

# What a Safety Computer is

**Safety computer**



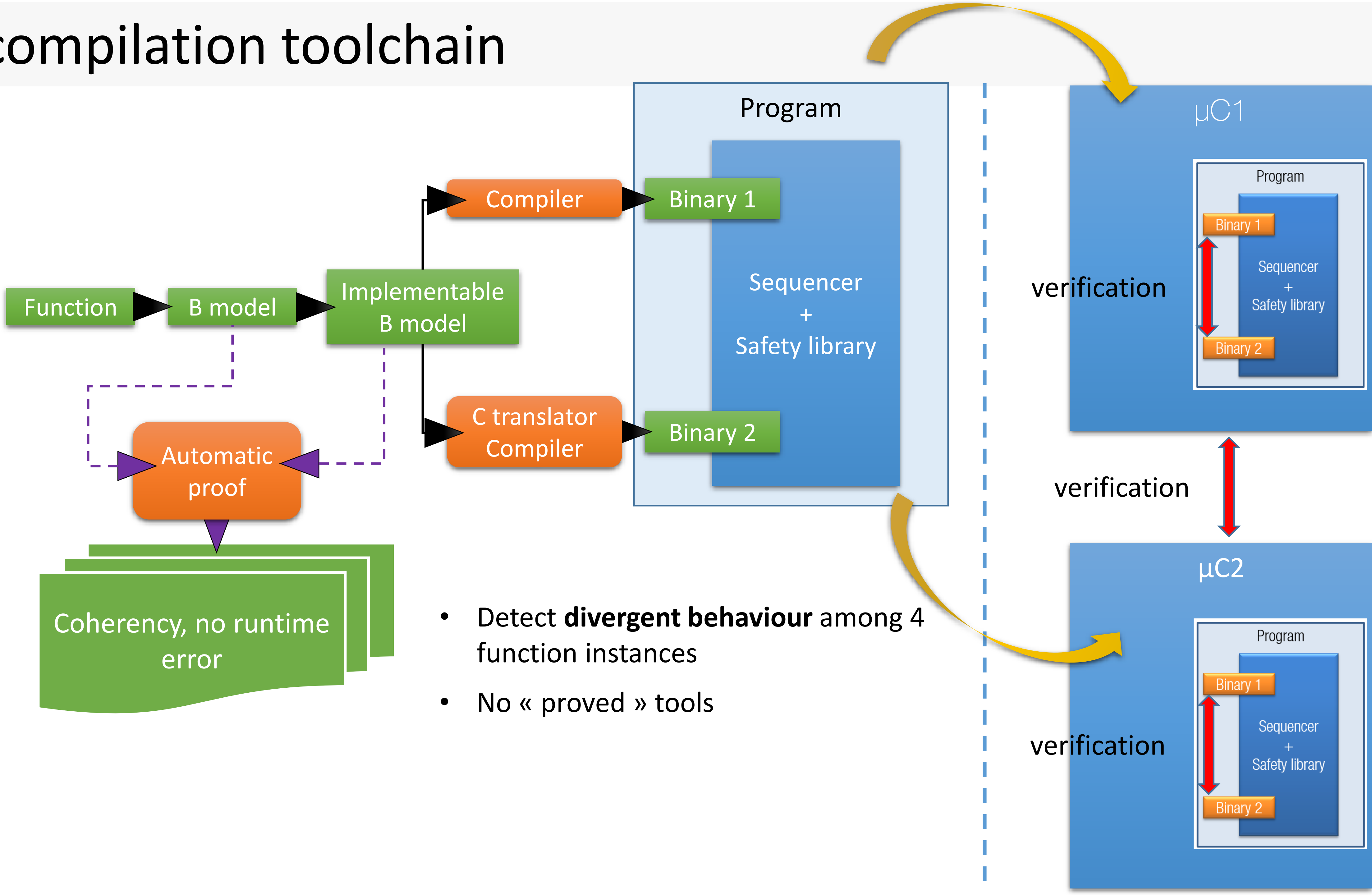

- **F == (read inputs, compute, set outputs)***
- **F** could harm / kill people
- Ability to check if able to execute **F** properly


*: for space applications, could be a reset when computer hit by high energy particle

- F is not safe just because a safety computer is used

***« execute the right F and execute the F right »***

# Double compilation toolchain




- Detect **divergent behaviour** among 4 function instances
- No « proved » tools

# CLEARSY Safety Platform

- Safety on hardware
  - Based on 2oo2 PIC32 microcontrollers
  - Offers up to 40 MIPS for lightweight applications

- Safety on software
  - Based on 4oo4 software
  - Correctness is ensured by mathematical proof (-> B method)
  - Cross checks between software instances and between microcontrollers


- Industrial software tools
- Based on Atelier B version 4.5 – Industrial Formal method
- Includes specific plugins to compile and load automatically to the platform.

# Verification

**Safety is built-in, out of reach of the developer who cannot alter it**

If one verification fails when loading or executing

- Bad CRC when bootloading code
- Bad memory map (overlap) when bootload
- $CRC(data_{Binary1}) \neq CRC(data_{Binary2})$ during execution on one µC
- Failing µC unable to handshake every 50 ms with other µC
- $CRC(data_{Binary})$ different on each µC (inter µC verification)
- Wrong input (absence of/incorrect sinusoidal signal)
- Outputs are not commandable
- **Output is ON when both µC agree**
- One µC is not able to execute properly instructions
- $CRC_{computed}$ (code) $\neq$ $CRC_{expected}$ (code) (deferred action)
- Etc.

Handle failures:

- **Systematic** (buggy code generator and compiler, etc.)
- **Random** (memory corruption, failing transistor, degrading clock, etc.)

Hyp

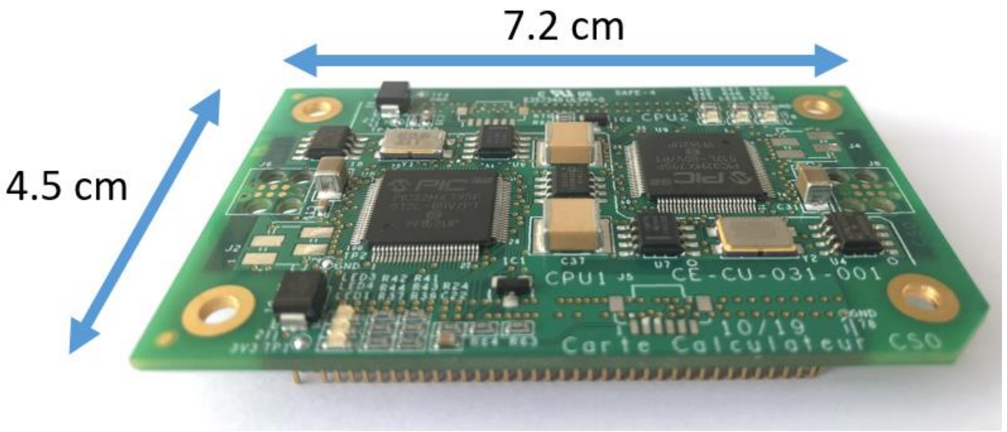
Models are proved to be correct:

- Syntax, types, properties
- No overflow, no division by 0, no access to a table outside of its bounds

# Available Boards

- **Starter kits for education**:
  - SK0 available since Q1 2019: 5 digital I/O
  - SK0 software simulator (no safety)
  - SK1 experimented in 2019: 28 digital I/O
  - SK2 (based on CS0) ETA 2022: 64 digital I/O


SK0 board

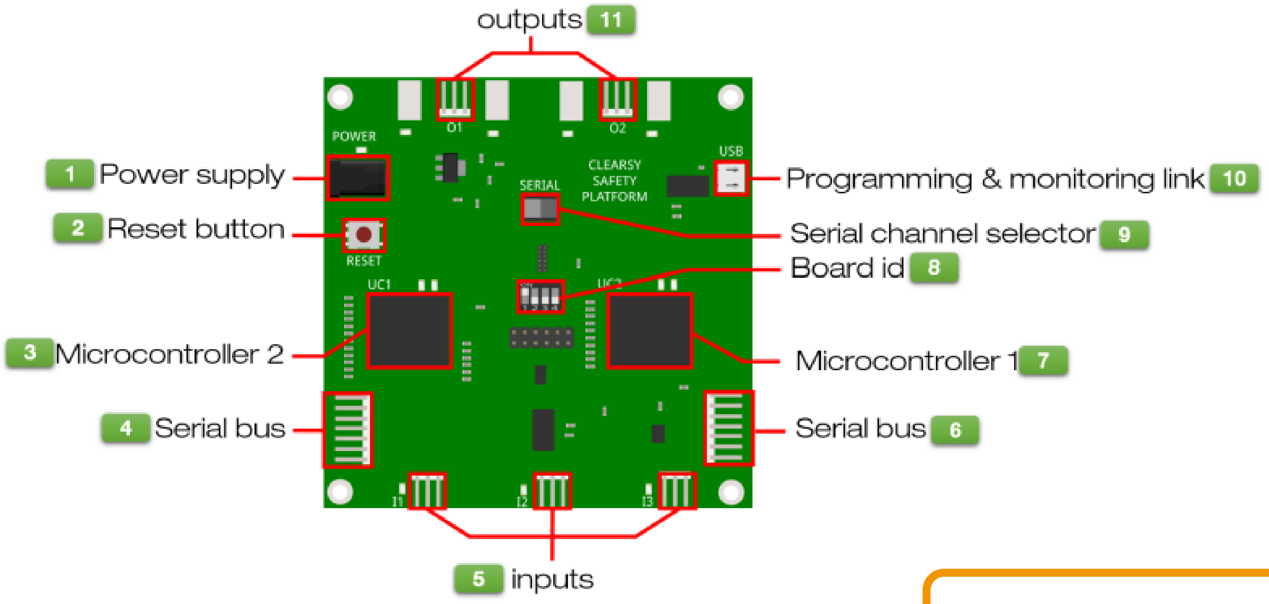


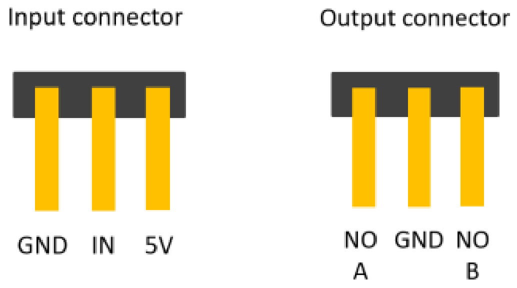

CS0 core computer

- **For industry (CS0 core computer)**
  - Certified SIL4
  - More flexibility
  - Programmed with B and C
  - Daughter board to be plugged on motherboard equipped with power supply and I/O

https://www.clearsy.com/en/our-tools/clearsy-safety-platform/

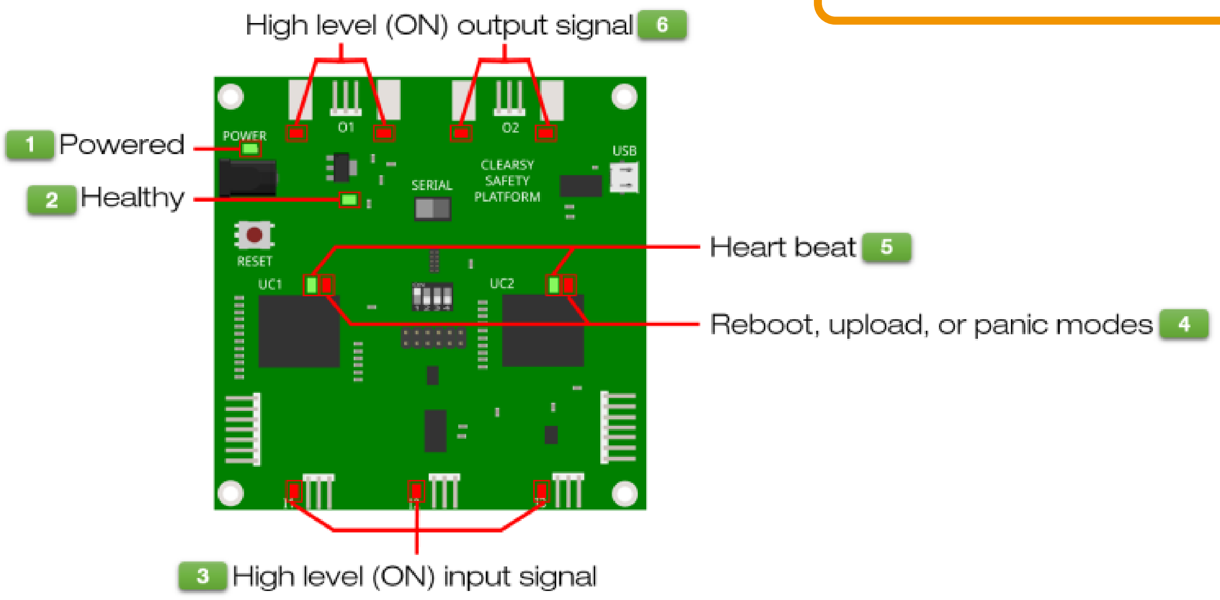
# Hardware interface



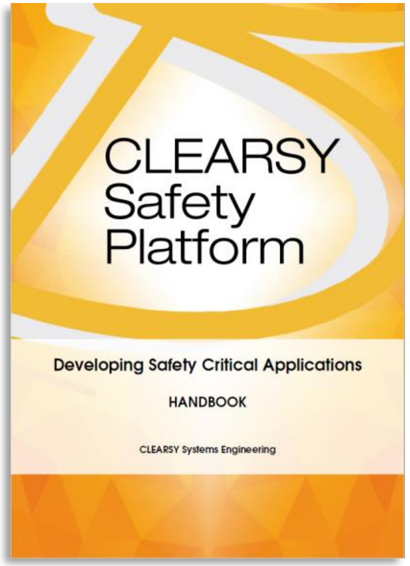

See §7. **Hardware interface** p51

https://www.clearsy.com/wp-content/uploads/2020/07/CSSP_User_Manual.pdf



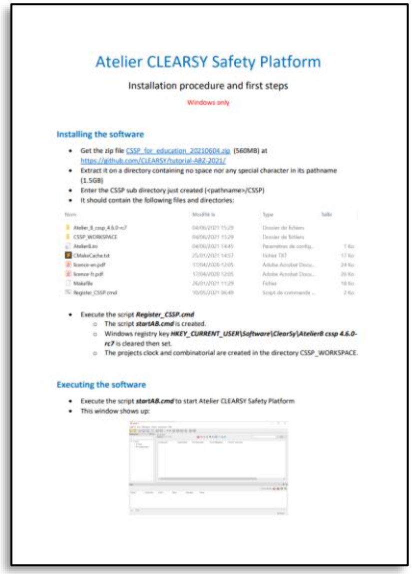

See §8. **LEDS** p55

# Useful links 1/2




- Handbook for software development

https://www.clearsy.com/en/download/download-documentation/




- CLEARSY Safety Platform IDE including SK0 software simulator

https://github.com/CLEARSY/tutorial-ABZ-2021/tree/main/Atelier%20CLEARSY%20Safety%20Platform

# Useful links 2/2

massive open online course

This course introduces the B-method: the basic concepts ranging from the most basic structures like the B machine to proofs using the Atelier-B interactive prover.

20+ hours vidéos

https://mooc.imd.ufrn.br/course/the-b-method

atelier B

BASIC & ADVANCED TRAINING

With these videos, you are going to be introduced to the tool and learn how to use it practically, for both software development and system modelling.

Introduction videos

https://www.youtube.com/playlist?list=PL2kYH179G4XJYeiznTe3t1axqYS7I0Nk8

```
INVARIANT
    arr:seq(AA) & nn = size(arr)
INITIALISATION arr,nn : (arr: seq(AA) & size(arr)=nn & nn:0..4)
OPERATIONS
    Reverse = VAR pp,qq IN
        pp,qq := 1,size(arr);
        WHILE pp<qq DO
            swap(arr,pp,qq);
            pp := pp+1;
            qq := qq-1
        INVARIANT pp:1..(1+size(arr)) & qq:0..size(arr)
        VARIANT 1+qq-pp
```

**A Collection of Formal Specifications** (University of Dusseldorf)
With this shared repository, B models from different origins are available for education and self-improvement.

B & Event-B models

https://github.com/hhu-stups/specifications/tree/master/prob-examples/B

# Summary

- CLEARSY Safety Platform
  - Safety computer
    - Execute 4 instances of the same function on 2 processors
    - Verify health regularly
    - Stop application execution and deactivate outputs if problem ❌
  - IDE
    - Application developed with B formal language
    - Model mathematically proved

# Agenda

Introduction to the CLEARSY Safety Platform

**Development process (demo video)**

Bits of B

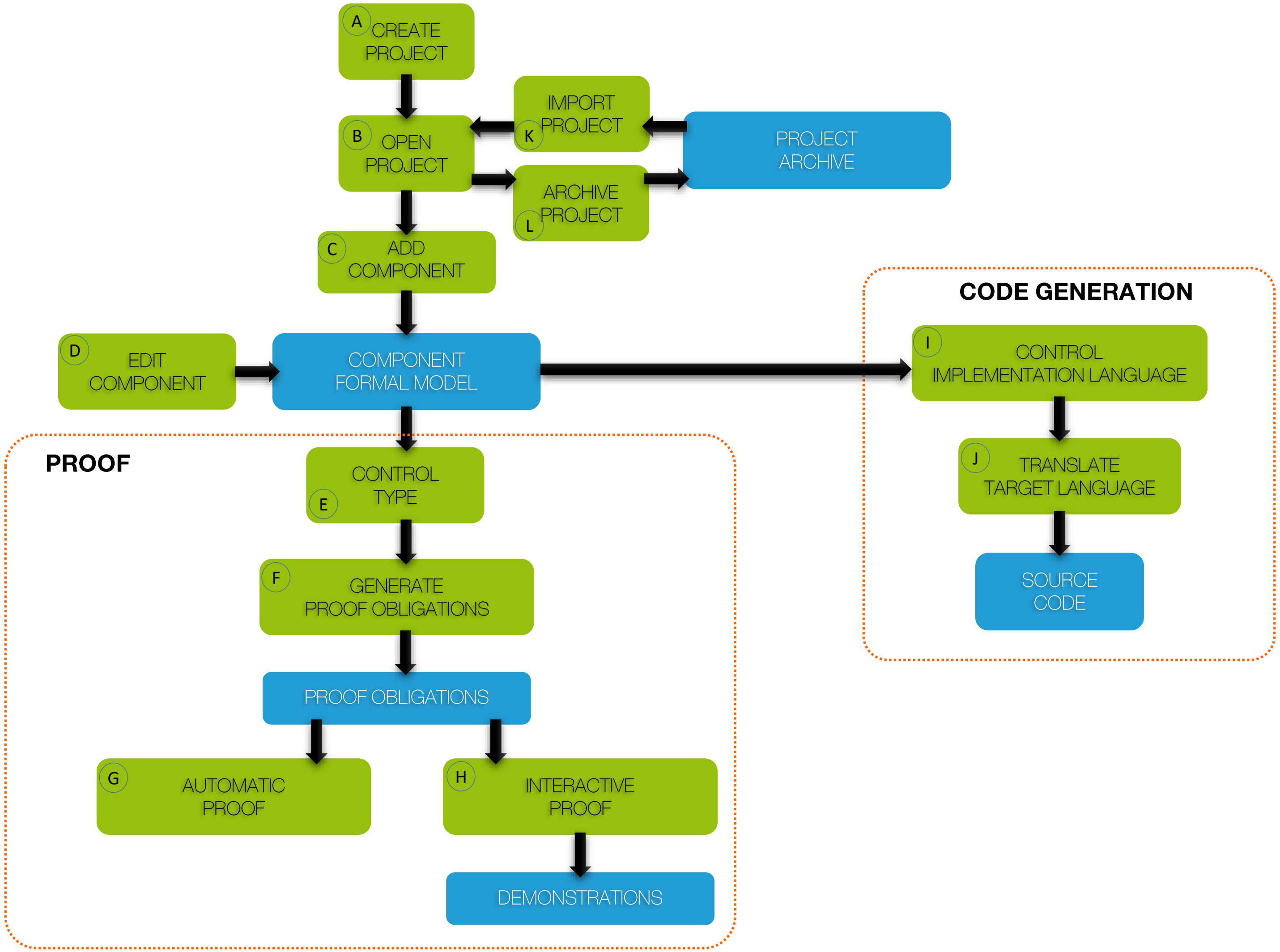
Using the modelling interface

The clock example (synchronous)

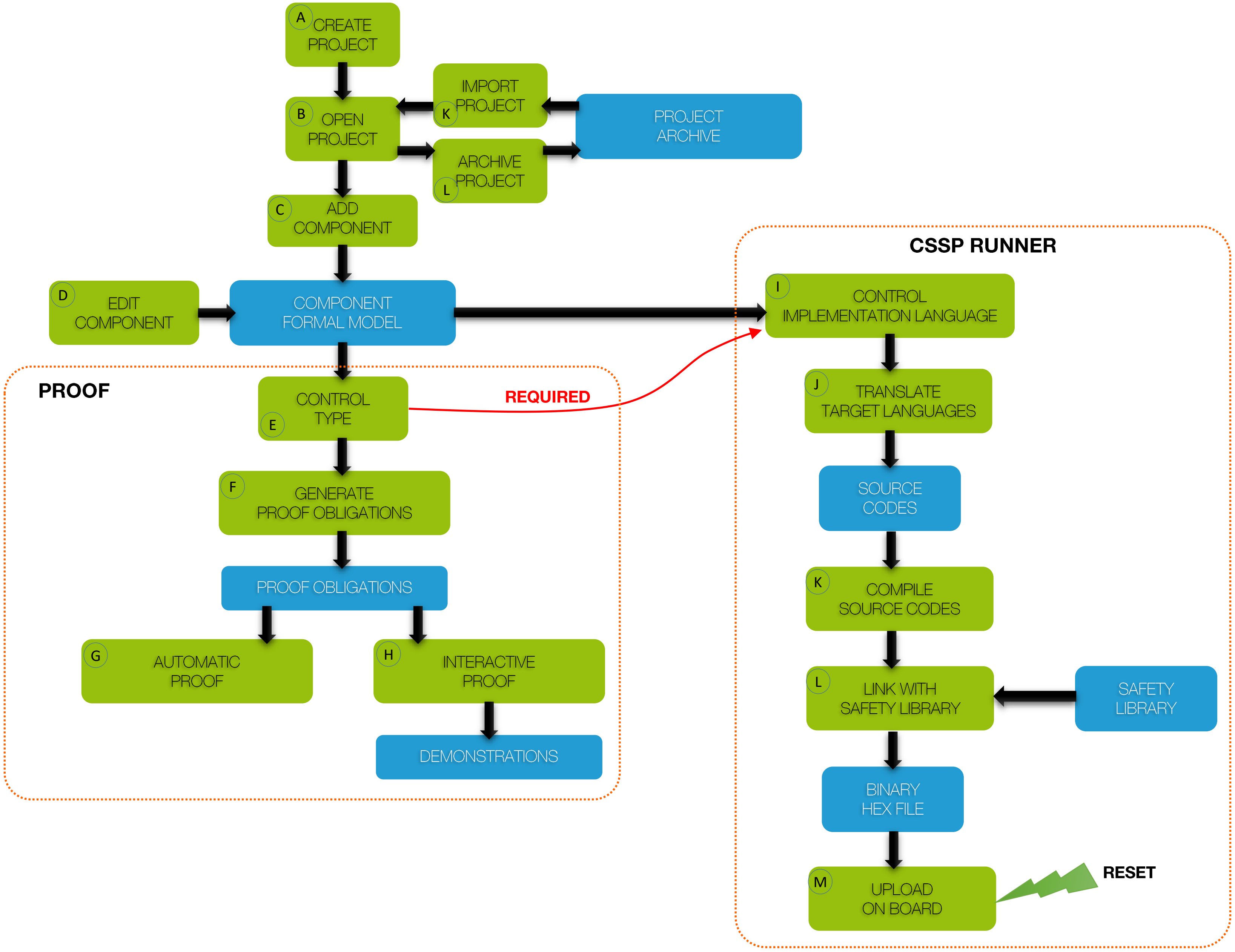
The combinatorial example (asynchronous)

Conclusion

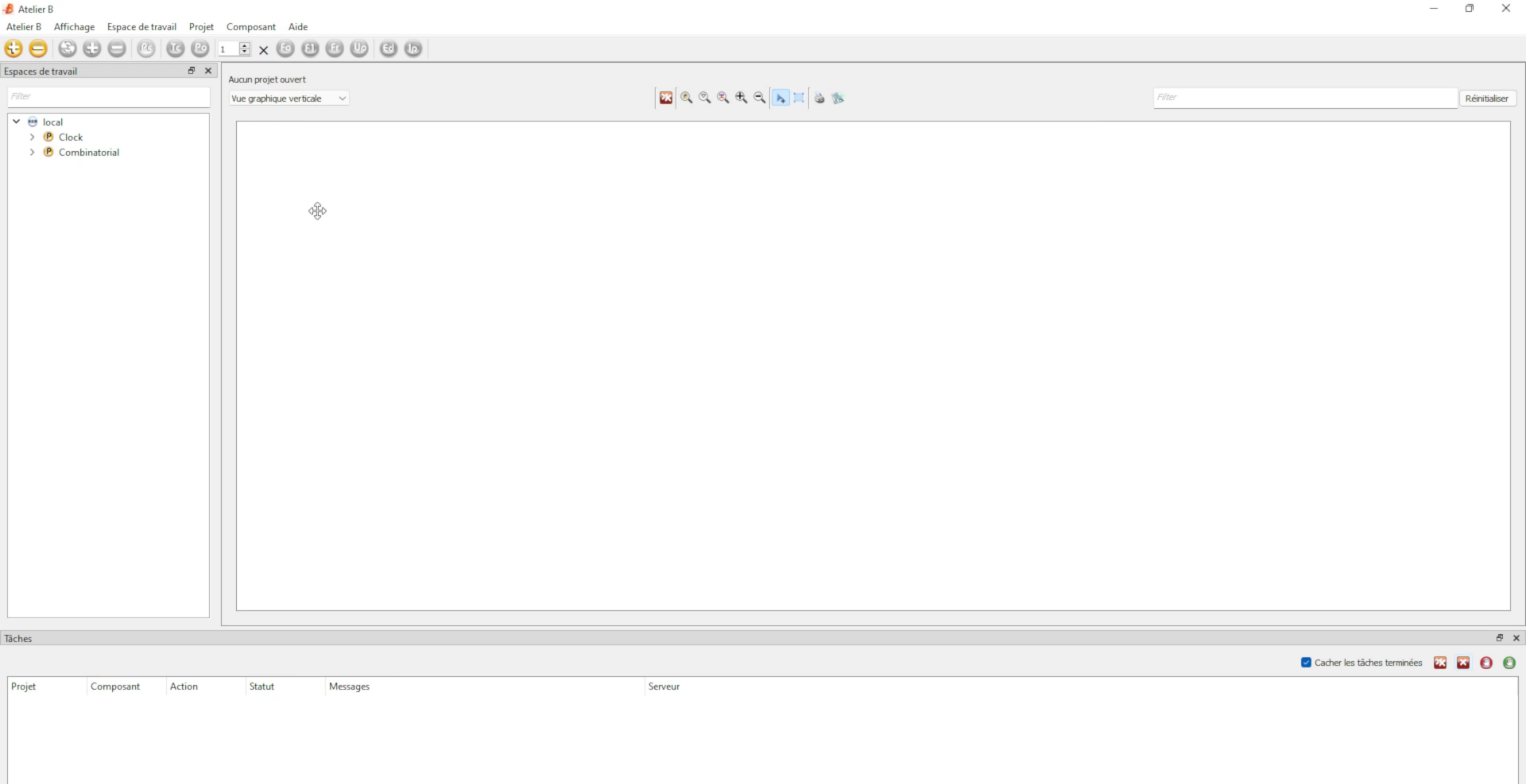
# Development Process (B)

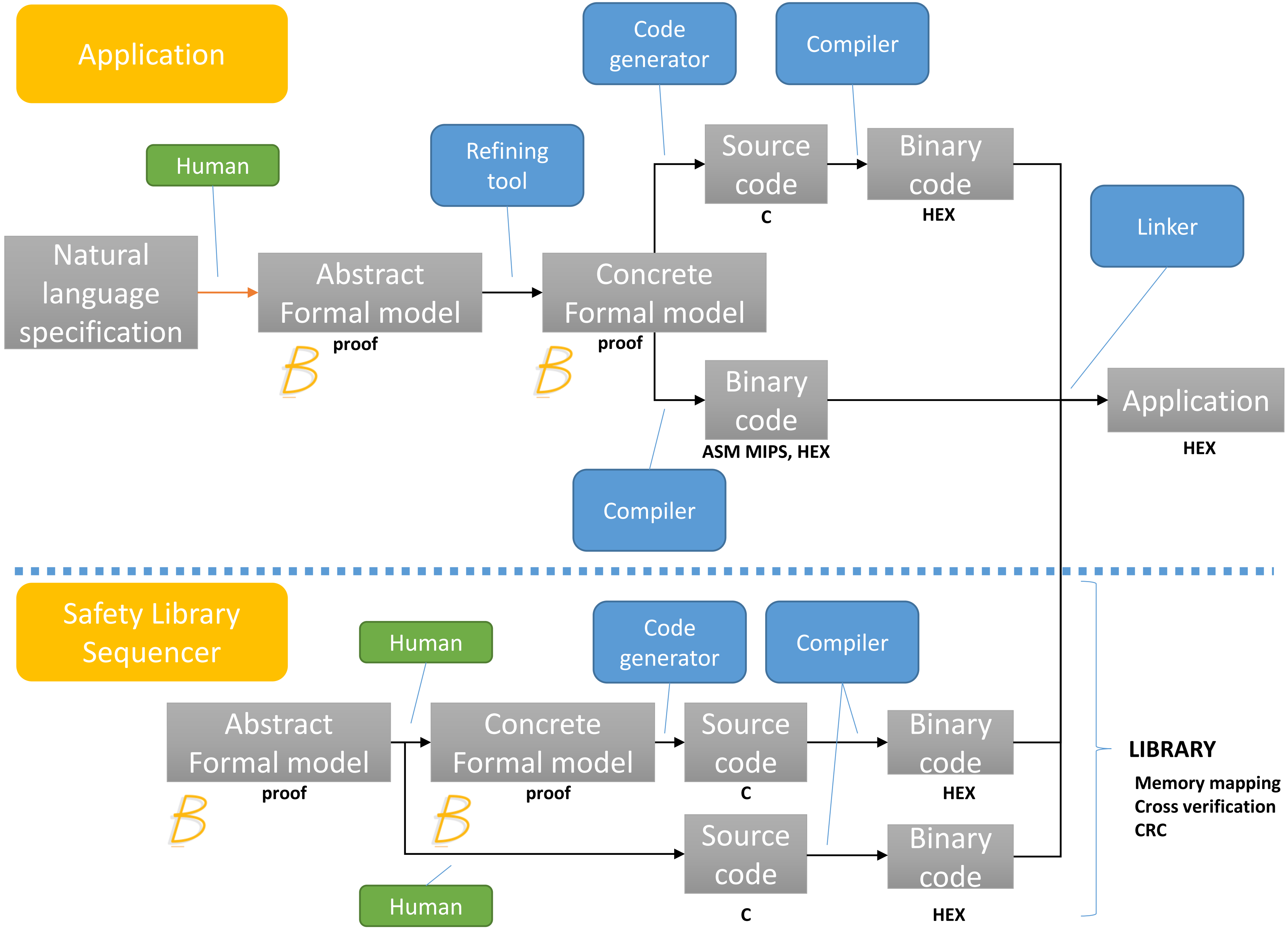


# Development Process (B+CSSP)



# Demo video


Atelier B
Atelier B
Affichage
Espace de travail
Projet
Composant
Aide
Espaces de travail
Filter
local
Clock
Combinatorial
Aucun projet ouvert
Vue graphique verticale
Filter
Réinitialiser
Tâches
Cacher les tâches terminées
Projet
Composant
Action
Statut
Messages
Serveur



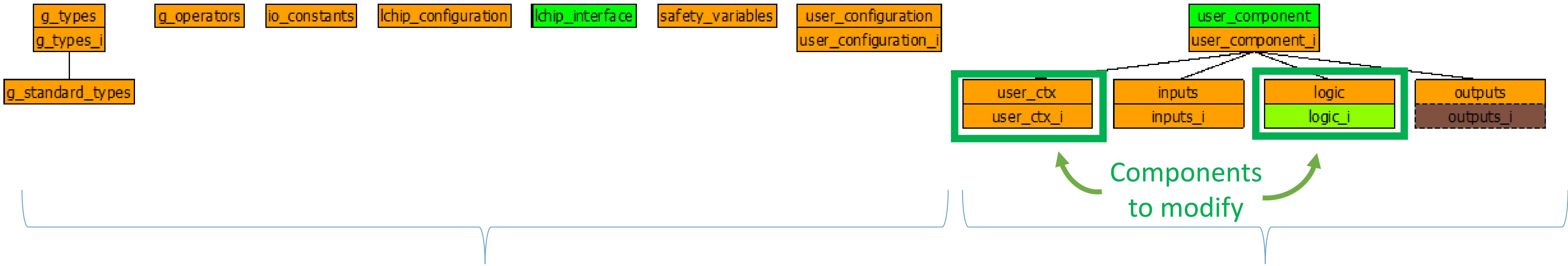
Application
Human
Refining tool
Code generator
Compiler
Natural language specification
Abstract Formal model
proof
Concrete Formal model
proof
Source code
C
Binary code
HEX
Linker
Binary code
ASM MIPS, HEX
Application
HEX
Compiler
Safety Library Sequencer
Human
Code generator
Compiler
Abstract Formal 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

# CSSP Project

## A CSSP project is a B project

- is generated automatically from board configuration (# IOs, naming)

## It contains




The interface with the safety library

The model of the function

# Programming Model & Applications

```
init();

while (1) {
   instance1();
   instance2();
}
```

- The execution is cyclic
- The function is executed regularly as often as possible similar to arduino programming (setup(), loop())

- No underlying operating system
- No interrupt()
- No predefined cycle time (if outputs are not set and cross read every **50ms**, board enters panic mode)
- No delay()
- Inputs are values captured at the beginning of a cycle (digital I/O)
- Outputs are maintained from one cycle to another (digital I/O)

- Project skeleton is generated from board description (I/O used, naming)
- Programming is specifying and implementing the function *user_logic*

# Summary

- CLEARSY Safety Platform
  - Safety computer
    - Program Flash with Runner after board reset
    - Execute program in Flash at startup
  - IDE
    - Atelier B
    - Redundant code generation

# Agenda

Introduction to the CLEARSY Safety Platform

Development process (demo video)

**Bits of B**

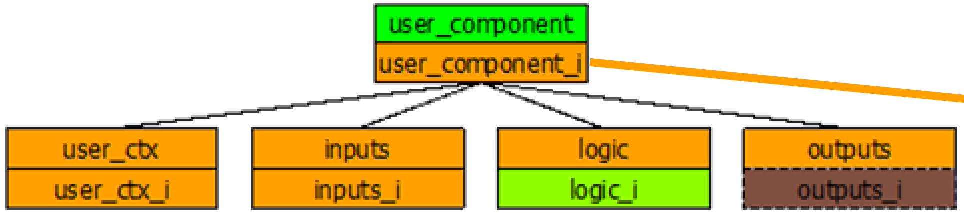
Using the modelling interface

The clock example (synchronous)

The combinatorial example (asynchronous)

Conclusion

# Generated Models




```
user_app =
BEGIN
    divergence_test_var :=  0;
    read_inputs;
    user_logic;
    write_outputs
END;
```

call

```
read_inputs  =
BEGIN
    I0 <-- read_global_input(0);
    I1 <-- read_global_input(1);
    I2 <-- read_global_input(2)
END;

po <-- get_I0 =
BEGIN
    po <-- read_global_input(0)
END;

po <-- get_I1 =
BEGIN
    po <-- read_global_input(1)
END;

po <-- get_I2 =
BEGIN
    po <-- read_global_input(2)
END
```

call

used by

```
user_logic = skip;

po <-- get_O0 =
BEGIN
    po := O0
END;

po <-- get_O1 =
BEGIN
    po := O1
END
```

call

used by

used by

```
write_outputs =
VAR
    lsb
IN
    lsb:(lsb : uint8_t);

    lsb <-- get_O0;
    write_global_output(0, lsb);

    lsb <-- get_O1;
    write_global_output(1, lsb)
END
```

Syntax:

**pp <-- ff(vv)**

represents a call to operation ff(vv)

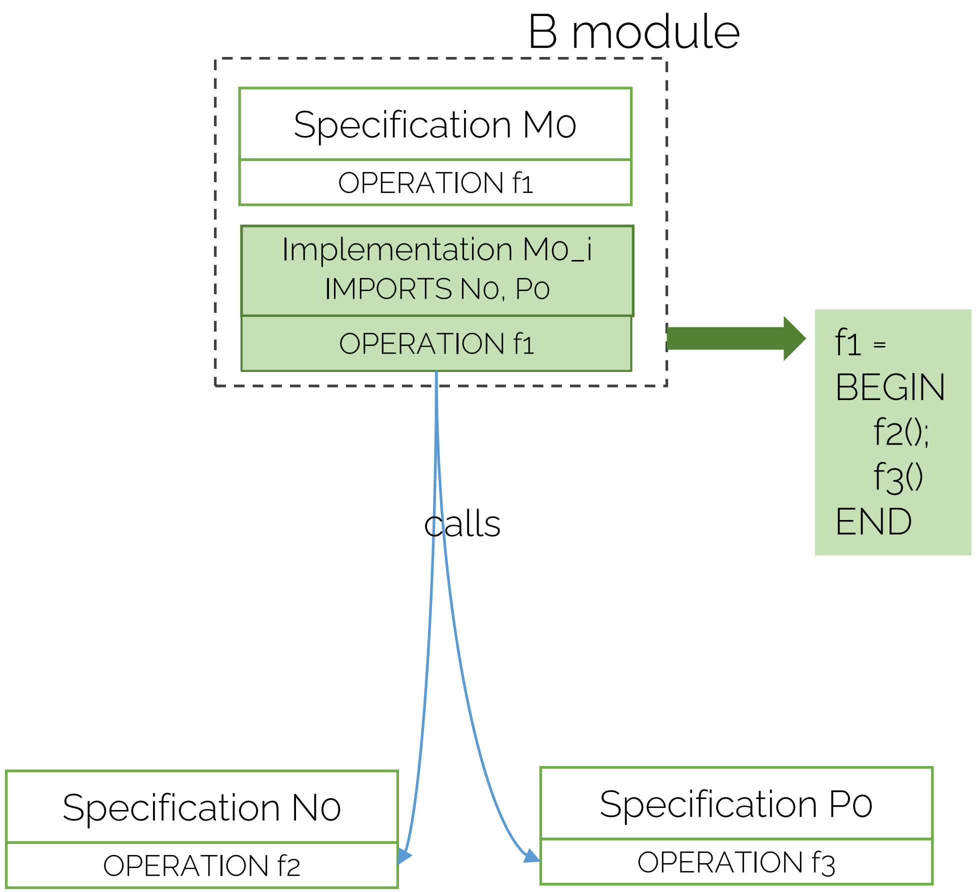
that returns the value pp

Programming CSSP is mainly implementing *user_logic* OPERATION

- Reading inputs values with *get_IO*, *get_I1* and *get_I2*
- Modifying the variables *O0* and *O1*

# Models architecture

- One operation cannot call other operations from the same implementation
- One operation can call operations defined in other machines
- These machines have to be imported in an implementation
- Variables defined in imported machines have to be different: a variable cannot be modified in 2 components

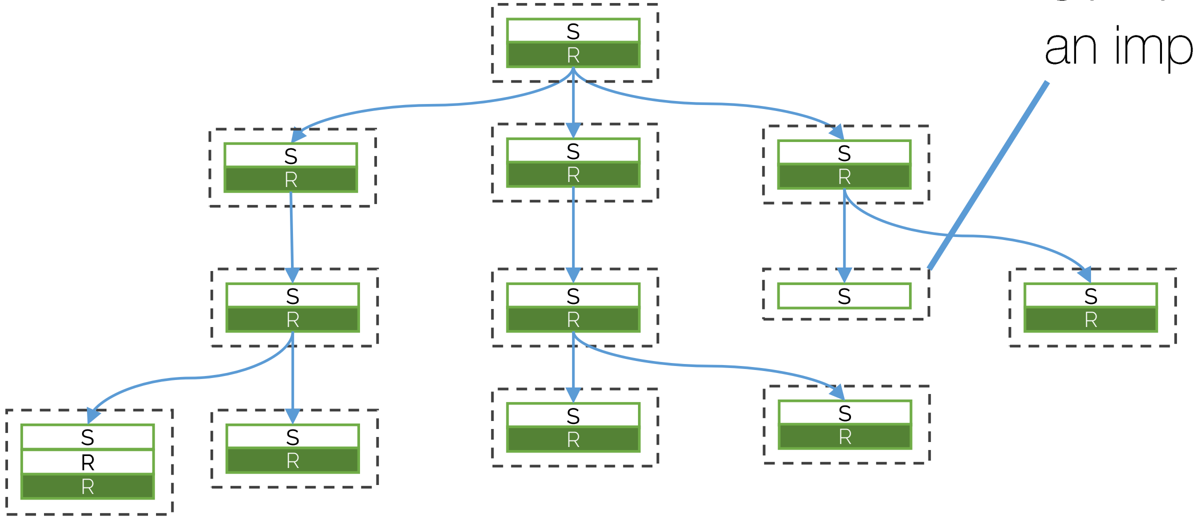


# Model Architecture

IMPORTS have to be applied iteratively to obtain the target decomposition

The decomposition graph should be a tree

Some components do not have an implementation in B



# B Variables Declaration

## specification

```
ABSTRACT_VARIABLES
    O0,
    O1
```

: means « belongs to »

```
INVARIANT
    O0 : uint8_t &
    O1 : uint8_t
```

|| means « in parallel », « at the same time »

```
INITIALISATION
    O0 :: uint8_t ||
    O1 :: uint8_t
```

:: means « any value within »

## implementation

```
// pragma SAFETY_VARS
```

**Mandatory**

Contains variables that will be verified

```
CONCRETE_VARIABLES
    O0,
    O1,
    TIME_A,
    STATUS
```

Variables local to implementation

```
INVARIANT
    O0 : uint8_t &
    O1 : uint8_t &
    TIME_A : uint32_t &
    STATUS : uint8_t

INITIALISATION
    O0 := IO_OFF;
    O1 := IO_OFF;
    TIME_A := 0;
    STATUS := SFALSE
```

# B Constants Declaration

## specification

```
CONCRETE_CONSTANTS
    DELTA_T
```

```
PROPERTIES
    DELTA_T : uint32_t
```

## implementation

```
// pragma CONSTANTS
```

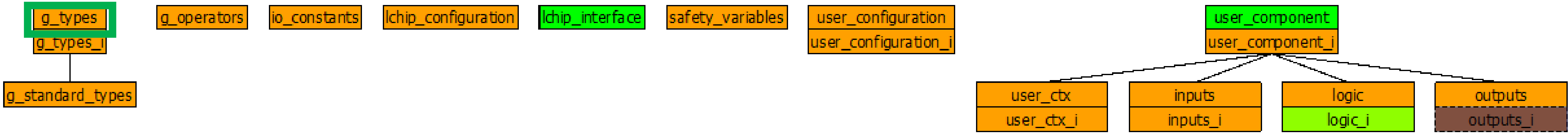
**Mandatory**
Contains constants that will be verified

**Important**
A model cannot contain both variables and constants

```
VALUES
    DELTA_T = 1000 // 1000 ms == 1s
```

Value that should enforce the properties

# CLEARSY Safety Platform Supported Types




```
CONCRETE_CONSTANTS
    uint32_t ,
    uint16_t ,
    uint8_t ,

    STRUE ,
    SFALSE ,
    MAX_UINT32 ,
    MAX_UINT16 ,
    MAX_UINT8
```

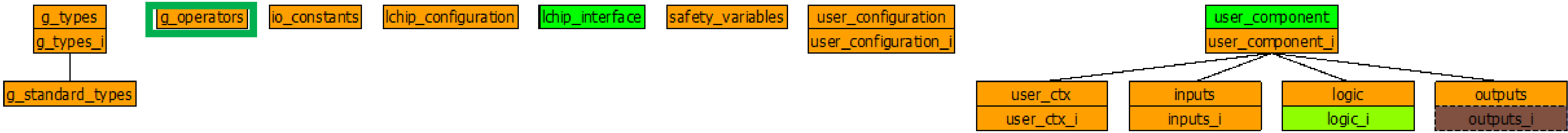
Everything is either 8, 16 or 32 bits

Boolean values TRUE and FALSE coded on 8 bits

The real values for STRUE and SFALSE are not displayed but we know that

```
PROPERTIES
    uint32_t = 0 .. 4294967295 &
    uint16_t = 0 .. 65535 &
    uint8_t  = 0 .. 255   &

    MAX_UINT32 : uint32_t &
    MAX_UINT16 : uint16_t &
    MAX_UINT8  : uint8_t  &

    STRUE  : uint8_t &
    SFALSE : uint8_t &

    MAX_UINT32 = 4294967295 &
    MAX_UINT16 = 65535      &
    MAX_UINT8  = 255   &

    STRUE  : 0 .. MAX_UINT8 &
    SFALSE : 0 .. MAX_UINT8 &

    STRUE /= SFALSE  &
    SBOOL = { STRUE , SFALSE } &
    STRUE <= 2 &
    SFALSE <= 2 &
```

# Unsigned INT Operators




```
CONCRETE_CONSTANTS
    bitwise_not_uint32,
    bitwise_and_uint32,
    bitwise_xor_uint32,
    bitwise_not_uint16,
    bitwise_and_uint16,
    bitwise_xor_uint16,
    bitwise_or_uint16,
    bitwise_not_uint8,
    bitwise_and_uint8,
    bitwise_xor_uint8,
    bitwise_or_uint8,
    add_uint32,
    sub_uint32,
    mul_uint32,
    add_uint16,
    sub_uint16,
    mul_uint16,
    add_uint8,
    sub_uint8,
    mul_uint8
```

Builtin operators

```
PROPERTIES
    bitwise_not_uint32 : uint32_t --> uint32_t &
    bitwise_and_uint32 : uint32_t * uint32_t --> uint32_t &
    bitwise_xor_uint32 : uint32_t * uint32_t --> uint32_t &
    bitwise_not_uint16 : uint16_t --> uint16_t &
    bitwise_and_uint16 : uint16_t * uint16_t --> uint16_t &
    bitwise_xor_uint16 : uint16_t * uint16_t --> uint16_t &
    bitwise_or_uint16  : uint16_t * uint16_t --> uint16_t &
    bitwise_not_uint8  : uint8_t --> uint8_t &
    bitwise_and_uint8  : uint8_t * uint8_t --> uint8_t &
    bitwise_xor_uint8  : uint8_t * uint8_t --> uint8_t &
    bitwise_or_uint8  : uint8_t * uint8_t --> uint8_t &
```

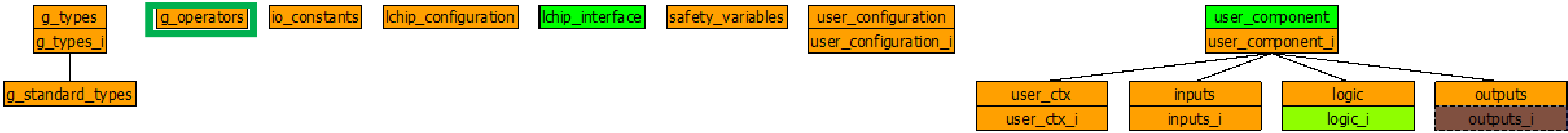
```
bitwise_not_uint32 : uint32_t --> uint32_t
```

total function that associates a 32-bit unsigned integer to any 32-bit unsigned integer

```
bitwise_and_uint32 : uint32_t * uint32_t --> uint32_t
```

total function with two 32-bit unsigned integer parameters

# Unsigned INT Operators



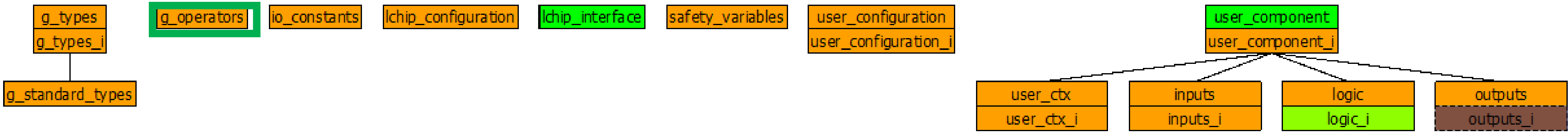

```
CONCRETE_CONSTANTS
    bitwise_not_uint32,
    bitwise_and_uint32,
    bitwise_xor_uint32,
    bitwise_not_uint16,
    bitwise_and_uint16,
    bitwise_xor_uint16,
    bitwise_or_uint16,
    bitwise_not_uint8,
    bitwise_and_uint8,
    bitwise_xor_uint8,
    bitwise_or_uint8,
    add_uint32,
    sub_uint32,
    mul_uint32,
    add_uint16,
    sub_uint16,
    mul_uint16,
    add_uint8,
    sub_uint8,
    mul_uint8
```

Builtin operators

```
add_uint32 : uint32_t * uint32_t --> uint32_t &
sub_uint32 : uint32_t * uint32_t --> uint32_t &
mul_uint32 : uint32_t * uint32_t --> uint32_t &
add_uint16 : uint16_t * uint16_t --> uint16_t &
sub_uint16 : uint16_t * uint16_t --> uint16_t &
mul_uint16 : uint16_t * uint16_t --> uint16_t &
add_uint8 : uint8_t * uint8_t --> uint8_t &
sub_uint8 : uint8_t * uint8_t --> uint8_t &
mul_uint8 : uint8_t * uint8_t --> uint8_t &
```

Operators that could lead to overflow

# Unsigned INT Operators




```
add_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 + x2) mod (MAX_UINT32 + 1)) &
sub_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 - x2 + MAX_UINT32 + 1) mod (MAX_UINT32 + 1)) &
mul_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 * x2) mod (MAX_UINT32 + 1)) &
add_uint16 = % (y1, y2) . (y1 : uint16_t & y2 : uint16_t | (y1 + y2) mod (MAX_UINT16 + 1)) &
sub_uint16 = % (y1, y2) . (y1 : uint16_t & y2 : uint16_t | (y1 - y2 + MAX_UINT16 + 1) mod (MAX_UINT16 + 1)) &
mul_uint16 = % (y1, y2) . (y1 : uint16_t & y2 : uint16_t | (y1 * y2) mod (MAX_UINT16 + 1)) &
add_uint8 = % (y1, y2) . (y1 : uint8_t & y2 : uint8_t | (y1 + y2) mod (MAX_UINT8 + 1)) &
sub_uint8 = % (y1, y2) . (y1 : uint8_t & y2 : uint8_t | (y1 - y2 + MAX_UINT8 + 1) mod (MAX_UINT8 + 1)) &
mul_uint8 = % (y1, y2) . (y1 : uint8_t & y2 : uint8_t | (y1 * y2) mod (MAX_UINT8 + 1)) &
```

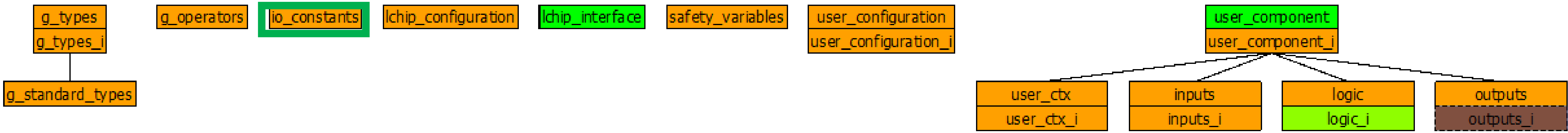
```
add_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 + x2) mod (MAX_UINT32 + 1))
```

is a λ function

that takes two 32-bit
unsigned integer parameters

and returns the sum of the values
modulo MAX_UINT32 +1

# Inputs / Outputs




```
ABSTRACT_CONSTANTS
    TIME,
    IO_STATE

CONCRETE_CONSTANTS
    IO_ON,
    IO_OFF

PROPERTIES
    TIME = uint32_t &
    IO_STATE = uint8_t &

    IO_ON  : uint8_t &
    IO_OFF : uint8_t &
    IO_ON /= IO_OFF &
    IO_ON : IO_STATE &
    IO_OFF : IO_STATE
```

inputs and outputs state

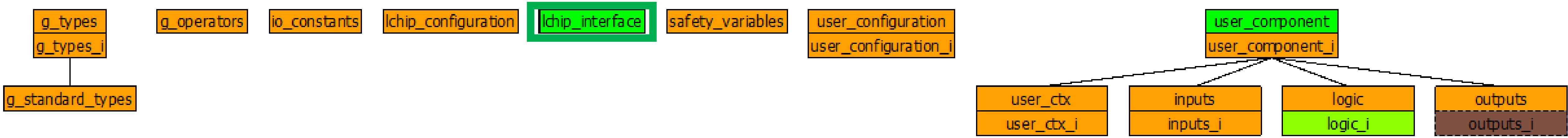
values used by digital inputs and outputs

coded on 8 bits

**Verification**

If a digital output is valued with a value different from IO_ON or IO_OFF then $SK_0$ stops in error mode

# Inputs / Outputs




```
out <-- get_ms_tick =
PRE
    out : uint32_t
THEN
    out := ms_tick
END
```

returns the number of milliseconds since the last reset

**Important**

$SK_0$ resets when the ms_tick reaches its upper bound i.e. every 49.7 days

# B Operations

Operations are populated with substitutions

Available substitutions in specification are different from the ones available in implementation

**specification**

Express the properties that the variables comply with <u>when the operation is completed</u> independently from the algorithm implemented (*post-condition*)

To simplify, always use « becomes such that substitutions »

```
user_logic  =
    BEGIN
        o0, o1 : (
            o0 : uint8_t &
            o1 : uint8_t &
            not(o0 = o1)
        )
    END;
```

Typing (mandatory)

Constraints (optional)

# B Operations

## implementation

```
user_logic = skip;
```

do nothing

```
user_logic =
BEGIN
    O0 := IO_ON;
    O1 := IO_OFF
END;
```

valuations in sequence

```
user_logic =
BEGIN
    IF Var8 = 0 THEN
        O0 := IO_ON
    ELSE
        O1 := IO_ON
    END
END;
```

IF THEN ELSE

**Important**

Only single condition (no conjonction nor disjonction)

= < <= operators only

```
user_logic =
BEGIN
    VAR time_ IN
        time_ : (time_ : uint32_t);
        time_ <-- get_ms_tick;
        IF 2000 <= time_ THEN
            O1 := IO_ON
        END
    END
END;
```

Local variables declaration

Operation call

**Important**

Local variables have to be typed first using « becomes such that » substitution

Contraints on the language to simplify the compiler

# user_logic

## specification

```
user_logic   = skip;
```

skip means « do no alter the variables of the model »

```
MACHINE
    logic

SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface

ABSTRACT_VARIABLES
    O1,
    O2

INVARIANT
    O1 : uint8_t &
    O2 : uint8_t

INITIALISATION
    O1 :: uint8_t ||
    O2 :: uint8_t
```

```
OPERATIONS
    user_logic   = skip;

    po <-- get_O1 =
    PRE
        po : uint8_t
    THEN
        po := O1
    END;

    po <-- get_O2 =
    PRE
        po : uint8_t
    THEN
        po := O2
    END
END
```

## implementation

```
user_logic   = skip;
```

Minimum example:

- do nothing; outputs remain in their initial state (INITIALISATION)

```
IMPLEMENTATION logic_i

REFINES logic

SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface,
    inputs

    // pragma SAFETY_VARS

CONCRETE_VARIABLES
    O1,
    O2

INVARIANT
    O1 : uint8_t &
    O2 : uint8_t
```

```
INITIALISATION
    O1 := IO_OFF;
    O2 := IO_OFF

OPERATIONS
    user_logic = skip;

    po <-- get_O1 =
    BEGIN
        po := O1
    END;

    po <-- get_O2 =
    BEGIN
        po := O2
    END
END
```

# user_logic

## specification

```
user_logic  =
BEGIN
    O0 :: uint8_t ||
    O1 :: uint8_t
END
```

O0 and O1 belong to their type

:( ) means « becomes such that »

```
user_logic  =
    BEGIN
        O0, O1 : (
            O0 : uint8_t &
            O1 : uint8_t &
            not(O0 = O1)
        )
    END
```

O0 and O1 belong to their type
and O0 is different from O1

```
user_logic  =
BEGIN
    O0 := IO_ON ||
    O1 := IO_OFF
END
```

Set O0 and reset O1

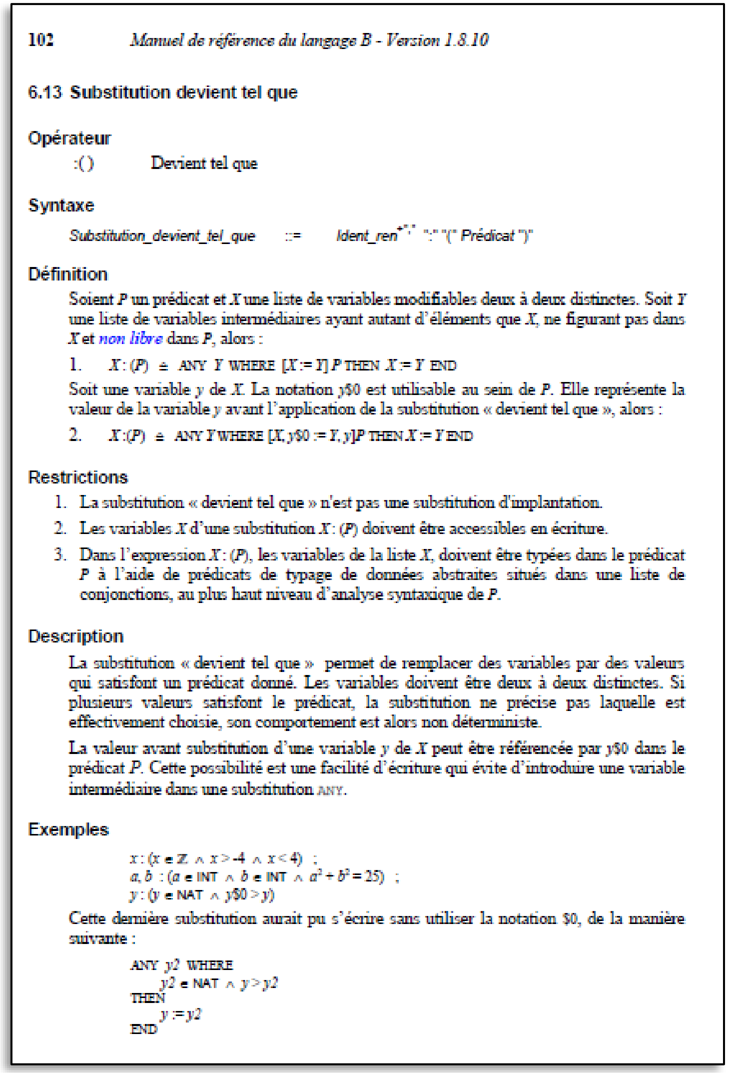
## implementation

```
user_logic =
BEGIN
    O0 := IO_ON;
    O1 := IO_OFF
END
```

Set O0 then reset O1

« then » is related to the valuation of O0 regarding O1
O0 and O1 will be positioned at the same time at the end of the cycle

# References



102 *Manuel de référence du langage B - Version 1.8.10*

**6.13 Substitution devient tel que**

**Opérateur**

:() Devient tel que

**Syntaxe**

*Substitution_devient_tel_que* ::= *Ident_ren*$^{+","}$ ":" "(" *Prédicat* ")"

**Définition**

Soient *P* un prédicat et *X* une liste de variables modifiables deux à deux distinctes. Soit *Y* une liste de variables intermédiaires ayant autant d'éléments que *X*, ne figurant pas dans *X* et *non libre* dans *P*, alors :

1. $X : (P) \;\hat{=}\;$ ANY $Y$ WHERE $[X := Y]\,P$ THEN $X := Y$ END

Soit une variable *y* de *X*. La notation *y*\$0 est utilisable au sein de *P*. Elle représente la valeur de la variable *y* avant l'application de la substitution « devient tel que », alors :

2. $X :(P) \;\hat{=}\;$ ANY $Y$ WHERE $[X, y\$0 := Y, y]P$ THEN $X := Y$ END

**Restrictions**

1. La substitution « devient tel que » n'est pas une substitution d'implantation.
2. Les variables *X* d'une substitution *X* : (*P*) doivent être accessibles en écriture.
3. Dans l'expression *X* : (*P*), les variables de la liste *X*, doivent être typées dans le prédicat *P* à l'aide de prédicats de typage de données abstraites situés dans une liste de conjonctions, au plus haut niveau d'analyse syntaxique de *P*.

**Description**

La substitution « devient tel que » permet de remplacer des variables par des valeurs qui satisfont un prédicat donné. Les variables doivent être deux à deux distinctes. Si plusieurs valeurs satisfont le prédicat, la substitution ne précise pas laquelle est effectivement choisie, son comportement est alors non déterministe.

La valeur avant substitution d'une variable *y* de *X* peut être référencée par *y*\$0 dans le prédicat *P*. Cette possibilité est une facilité d'écriture qui évite d'introduire une variable intermédiaire dans une substitution ANY.

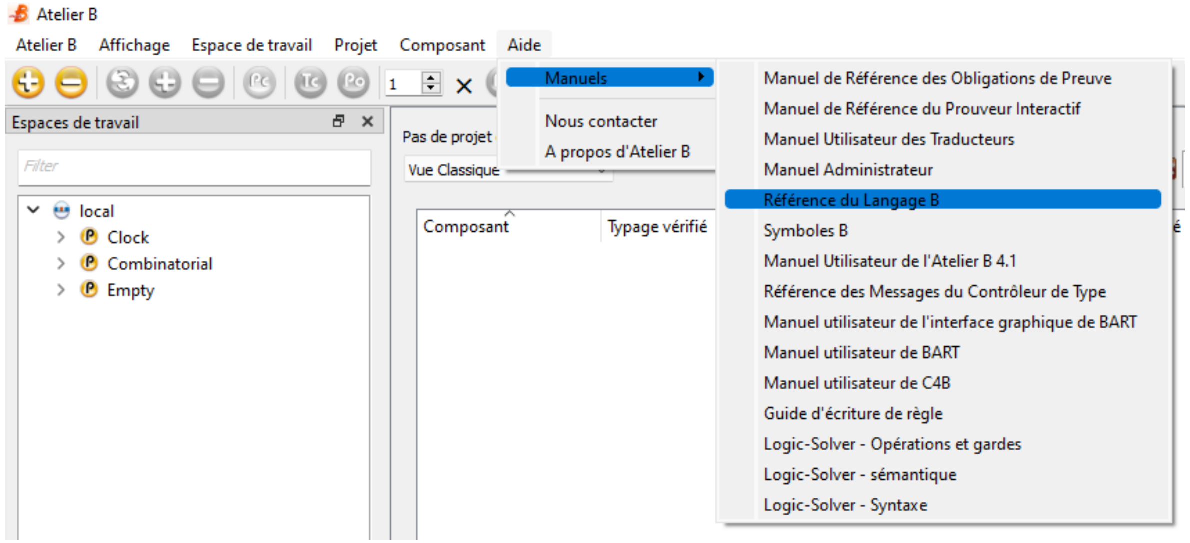
**Exemples**

$x : (x \in \mathbb{Z} \wedge x > -4 \wedge x < 4)$ ;
$a, b : (a \in \mathrm{INT} \wedge b \in \mathrm{INT} \wedge a^2 + b^2 = 25)$ ;
$y : (y \in \mathrm{NAT} \wedge y\$0 > y)$

Cette dernière substitution aurait pu s'écrire sans utiliser la notation \$0, de la manière suivante :

```
ANY y2 WHERE
    y2 ∈ NAT ∧ y > y2
THEN
    y := y2
END
```

- B Language Reference Manual in Atelier B




- Handbook for software development
  - B language restrictions

# Summary

- CLEARSY Safety Platform
  - Support
    - restricted B language,
    - Specific types and operators
    - Specific syntax including pragmas

# Agenda

Introduction to the CLEARSY Safety Platform

Development process (demo video)

Bits of B

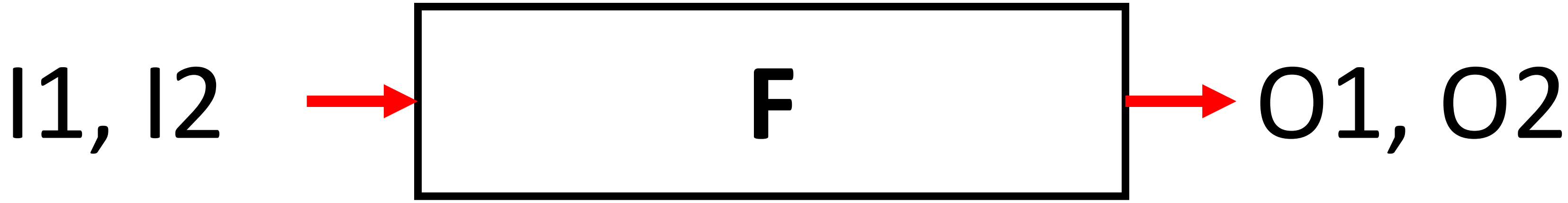
**Using the modelling interface**

The clock example (synchronous)

The combinatorial example (asynchronous)

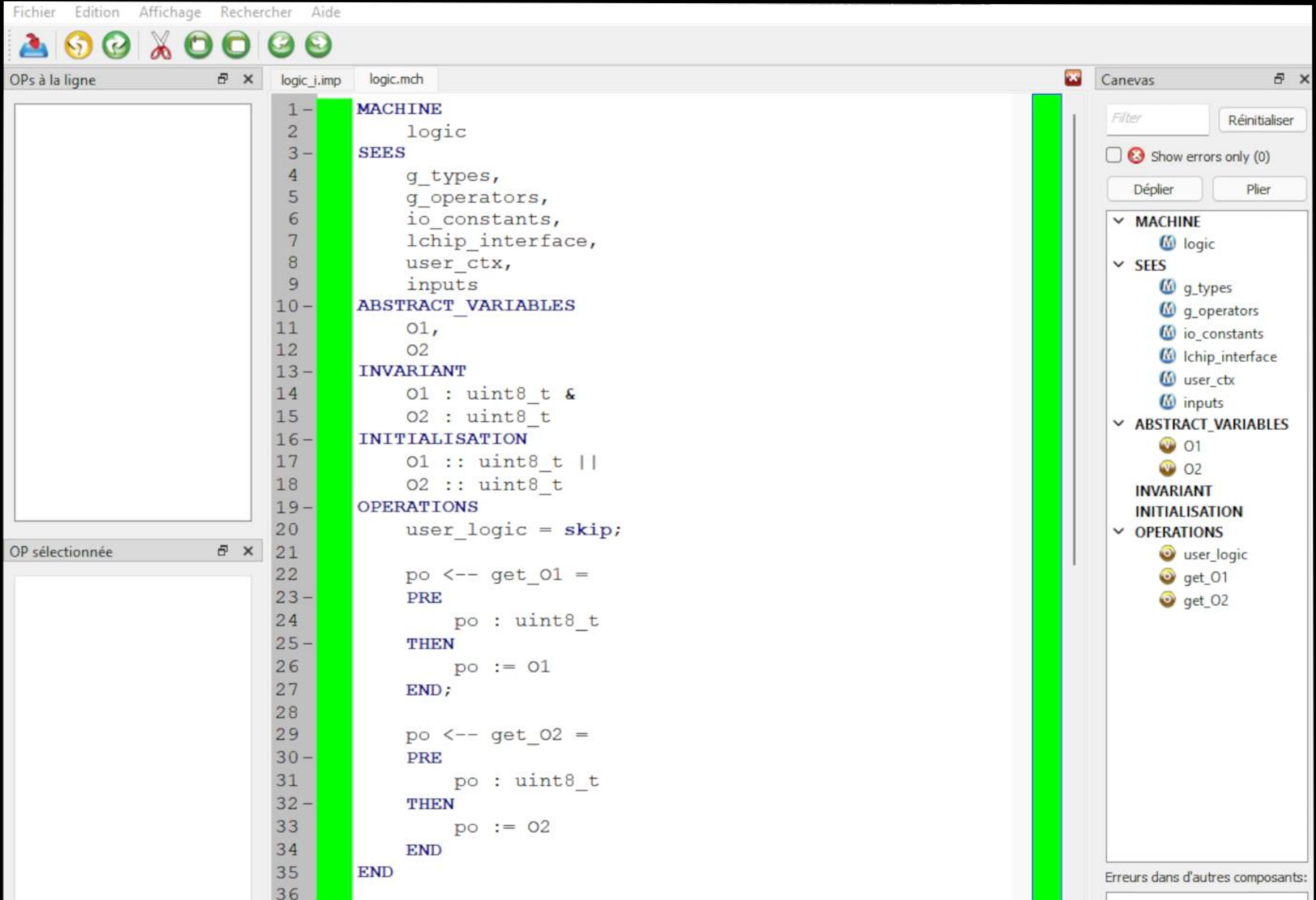
Conclusion

# Example (specification)




- I1, I2, O1, O2 belongs to {IO_OFF, IO_ON}
- O1 is IO_ON *iff* both I1 and I2 are IO_ON
- O2 is the complement of O1

# Example (specification)



# Example (specification)

I1, I2 → **F** → O1, O2

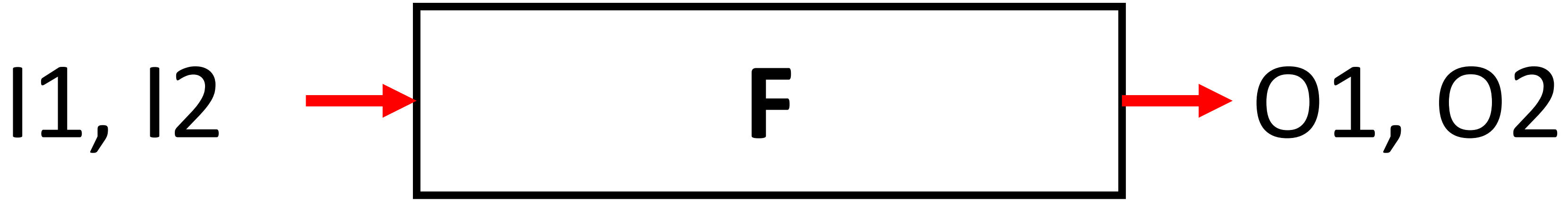
- I1, I2, O1, O2 belongs to {IO_OFF, IO_ON} is unsigned 8 bit integer enumeration
- O1 is IO_ON *iff* both I1 and I2 are IO_ON
- O2 is the complement of O1

```
user_logic =
BEGIN
    o1, o2: (
        o1 : uint8_t &
        o2 : uint8_t
    )
END;
```

« 01, 02 become such that
- Type of O1 is unsigned 8 bit integer
- Type of O2 is unsigned 8 bit integer »

Minimum specification :« 01 and 02 are modified in accordance with their type »

# Example (specification)




- I1, I2, O1, O2 belongs to {IO_OFF, IO_ON}
- O1 is IO_ON *iff* both I1 and I2 are IO_ON
- O2 is the complement of O1

```
user_logic =
BEGIN
    O1, O2: (
        O1 : uint8_t &
        O2 : uint8_t &
        (O1=IO_ON <=> (I1=IO_ON & I2=IO_ON)) &
        not(O1 = O2)
    )
END
```

# Example (implementation)

I1, I2 → **F** → O1, O2

Green means « Implementation Fully proved Against Specification »

```
user_logic =
BEGIN
    VAR i1_, i2_ IN
        i1_ : (i1_ : uint8_t);
        i2_ : (i2_ : uint8_t);

        i1_ <-- get_I1;
        i2_ <-- get_I2;

        O1 := IO_OFF;
        O2 := IO_ON;
        IF i1_ = IO_ON THEN
            IF i2_ = IO_ON THEN
1/1             O1 := IO_ON;
1/1             O2 := IO_OFF
            END
        END
    END
END
```

Get I1 and I2 values

# Example (code generation)

```
user_logic =
BEGIN
    VAR i1_ , i2_ IN
        i1_ : (i1_ : uint8_t);
        i2_ : (i2_ : uint8_t);

        i1_ <-- get_I1;
        i2_ <-- get_I2;

        O1 := IO_OFF;
        O2 := IO_ON;
        IF i1_ = IO_ON THEN
            IF i2_ = IO_ON THEN
                O1 := IO_ON;
                O2 := IO_OFF
            END
        END
    END
END
```

```
void SECTION_C4B_FUNCTION user_logic(void)
{
    {
        uint8_t i1_;
        uint8_t i2_;

        get_I1(&i1_);
        get_I2(&i2_);
        O1 = IO_OFF;
        O2 = IO_ON;
        if(i1_ == IO_ON)
        {
            if(i2_ == IO_ON)
            {
                O1 = IO_ON;
                O2 = IO_OFF;
            }
        }
    }
}
```

**Path** *AB_CSSP_4.6.0-RC7\Example\lang\dcc\Example\c_toolchain_1\sources\replica_2*

# Summary

- CLEARSY Safety Platform
  - Edit logic.mch
  - Edit logic_i.imp
  - Save to check if the model
    - has a correct type
    - is proved

# Agenda

Introduction to the CLEARSY Safety Platform

Development process (demo video)
Bits of B
Using the modelling interface

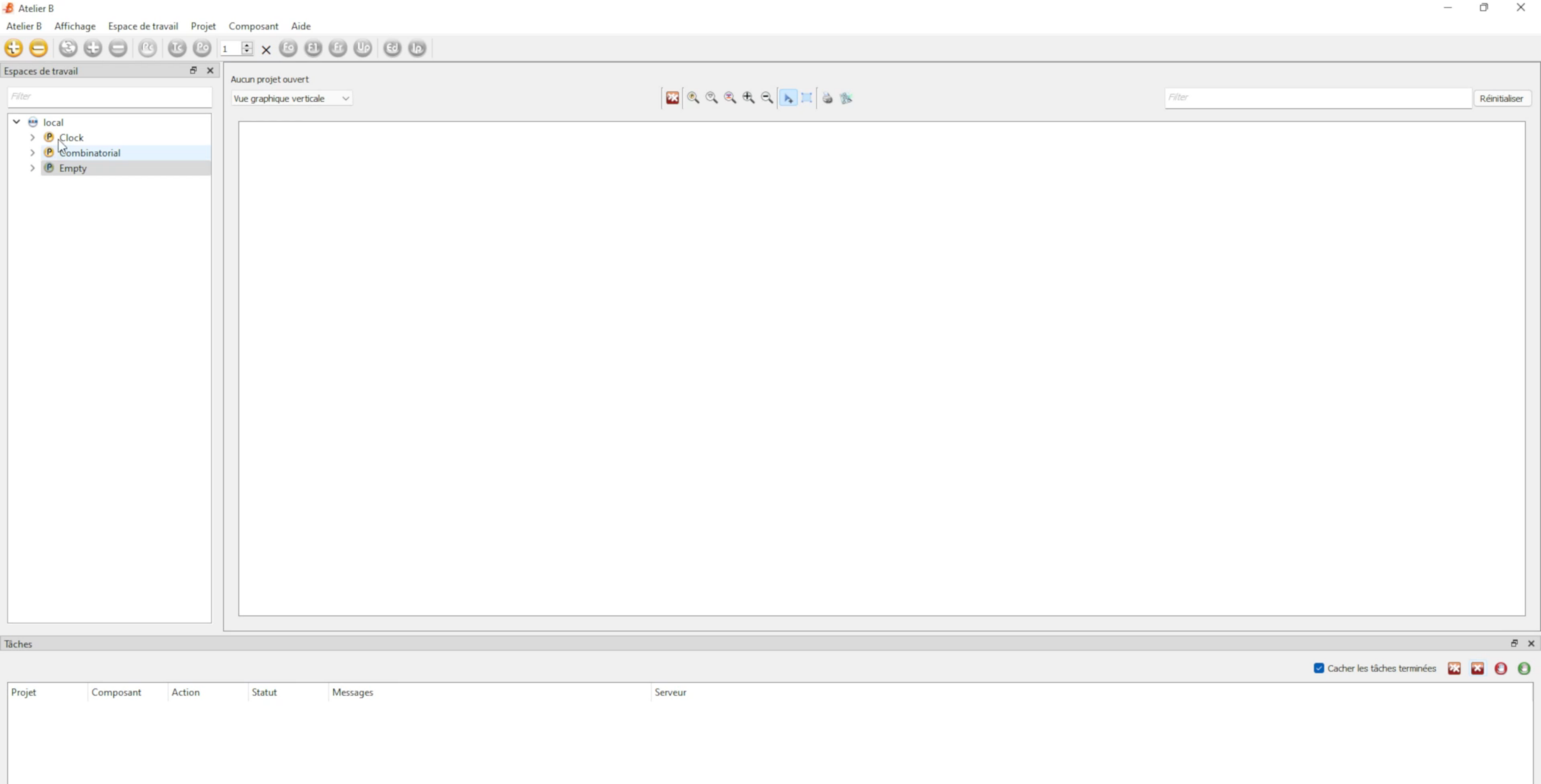
**The clock example (synchronous)**
The combinatorial example (asynchronous)

Conclusion

# Clock example

$O_1 = \text{not}(O_1)$ every 1 second
$O_2 = \text{not}(O_1)$

# Clock example



# Clock example

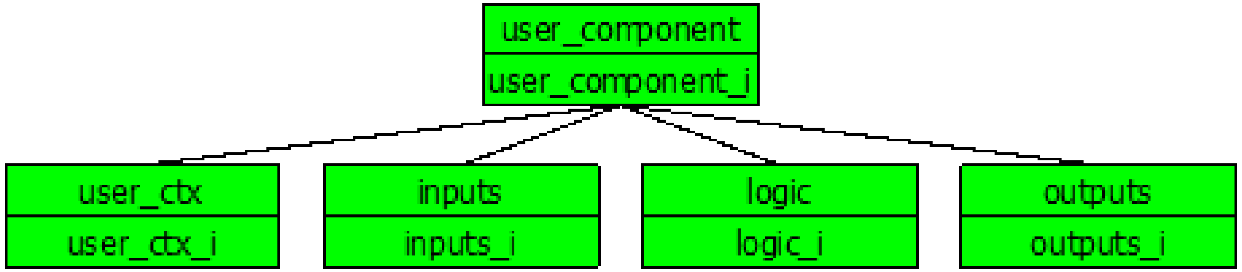
$O_1 = \text{not}(O_1)$ every 1 second
$O_2 = \text{not}(O_1)$

Current time

User defined constant

```
user_logic =
BEGIN
    VAR ltime_, s_tick_cycle, status_ IN
        ltime_ :(ltime_ : uint32_t);
        s_tick_cycle :(s_tick_cycle : uint32_t);
        status_ :(status_ : uint32_t);

        ltime_ <-- get_ms_tick;
        s_tick_cycle := ltime_ / DELTA_T;
        status_ := s_tick_cycle mod 2;

        O2 := IO_OFF;
        IF status_ = 0 THEN
            O2 := IO_ON
        END;


        IF O2 = IO_ON THEN
            O1 := IO_OFF
        ELSE
            O1 := IO_ON
        END
    END
END;
```

# Clock example




```
MACHINE
    user_ctx
SEES
    g_types
CONCRETE_CONSTANTS
    DELTA_T
PROPERTIES
    DELTA_T : uint32_t
END
```

Read access to g_types for uint32_t declaration

```
IMPLEMENTATION
    user_ctx_i
REFINES
    user_ctx

    // pragma CONSTANTS
SEES
    g_types
VALUES
    DELTA_T = 1000 // 1000 ms == 1s
END
```

Contains constants

# Clock example

Your turn:

- **Model**:
  - Open the project clock
  - Have a look at the component logic / logic_i , OPERATION user_logic
- **Prove**:
  - Ctrl+A (component view) to select all components, press F0 to start type check, PO generation and proof in sequence
- **Compile**:
  - Right click on the project (left pane) then select « CSSP runner »
- **Upload**:
  - Connect your board, click on the green arrow, reset your board, wait for « device ready », reset your board
- **Check** that your output relays change state every second

# Going further

Your turn:

- **Program 2 clocks with different cycle time** (one on $O_1$, the other on $O_2$)

  $O_1$ = not($O_1$) every xxx milli-seconds

  $O_2$ = not($O_2$) every yyy milli-seconds

- **Do not change the status of the outputs too often (< 50 ms) or you will kill the relays !**

- Model, prove, compile, upload, reset your device
- Check with your buttons that the function is correctly implemented

# Agenda

Introduction to the CLEARSY Safety Platform

Development process (demo video)
Bits of B
Using the modelling interface

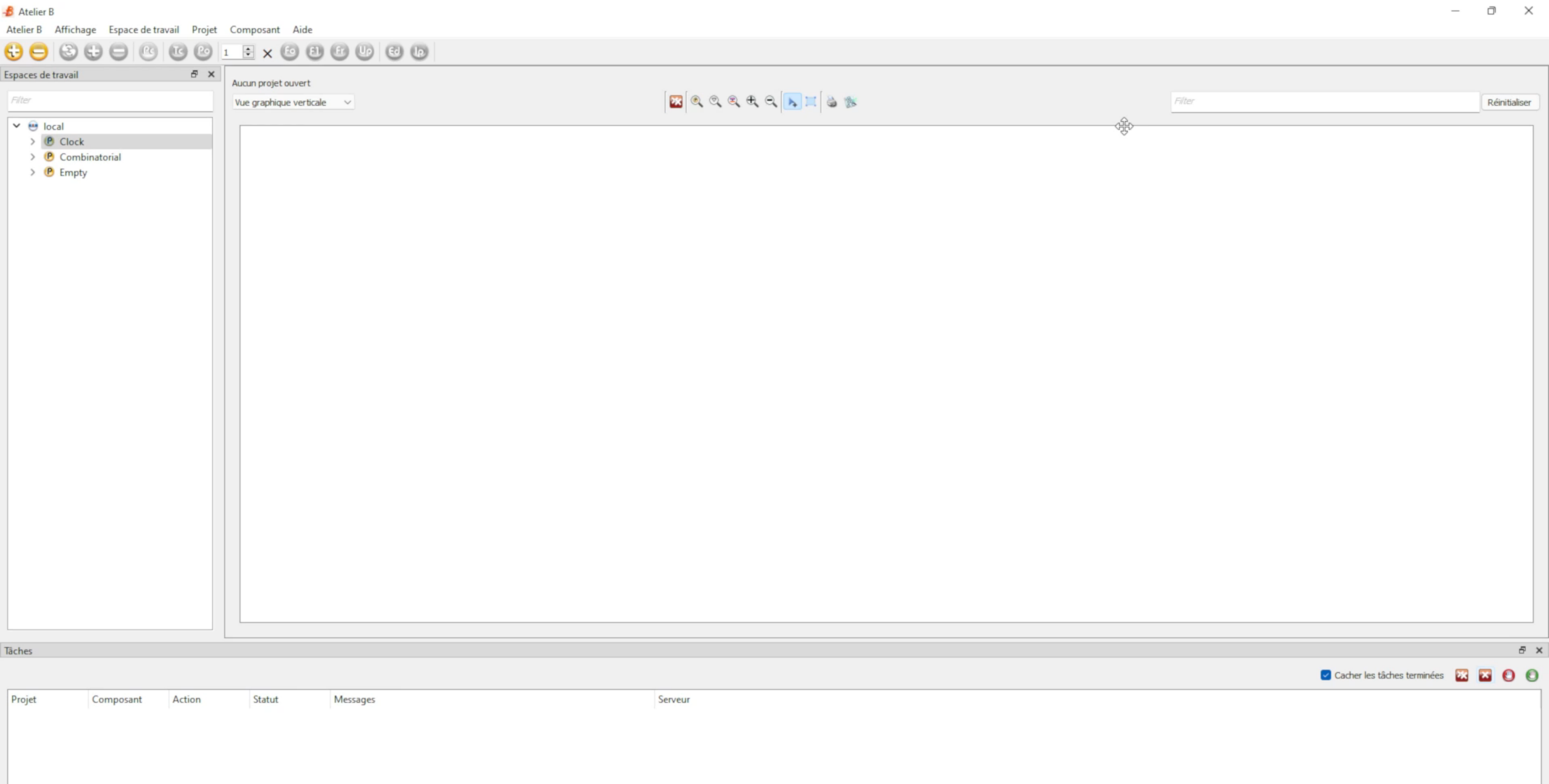
The clock example (synchronous)
**The combinatorial example (asynchronous)**

Conclusion

# Combinatorial example

$O_0 = I_0$ and $I_1$ and $I_2$
$O_1 = \text{not}(O_0)$

# Combinatorial example



# Combinatorial example

$O_0 = I_0$ and $I_1$ and $I_2$
$O_1 = \mathrm{not}(O_0)$

```
user_logic =
BEGIN
    VAR i0_, i1_, i2_ IN
        i0_ : (i0_ : uint8_t);
        i1_ : (i1_ : uint8_t);
        i2_ : (i2_ : uint8_t);

        i0_ <-- get_I0;
        i1_ <-- get_I1;
        i2_ <-- get_I2;

        O0 <-- triAND(i0_, i1_, i2_); /* O0 is ON iff I0, I1 & I2 are ON */
        O1 <-- negIO(O0) /* O1 is the opposite of O0 */
    END
END
;
```

Local variables are typed first

Local variables are valued

Variables are valued with LOCAL_OPERATIONS

# Combinatorial example

```
LOCAL_OPERATIONS
    res <-- triAND(v1, v2, v3) =
    PRE
        v1: uint8_t &  v2: uint8_t & v3: uint8_t
    THEN
        res :: uint8_t
    END
    ;
    res <-- negIO(val) =
    PRE
        val : uint8_t
    THEN
        res :: uint8_t
    END
```

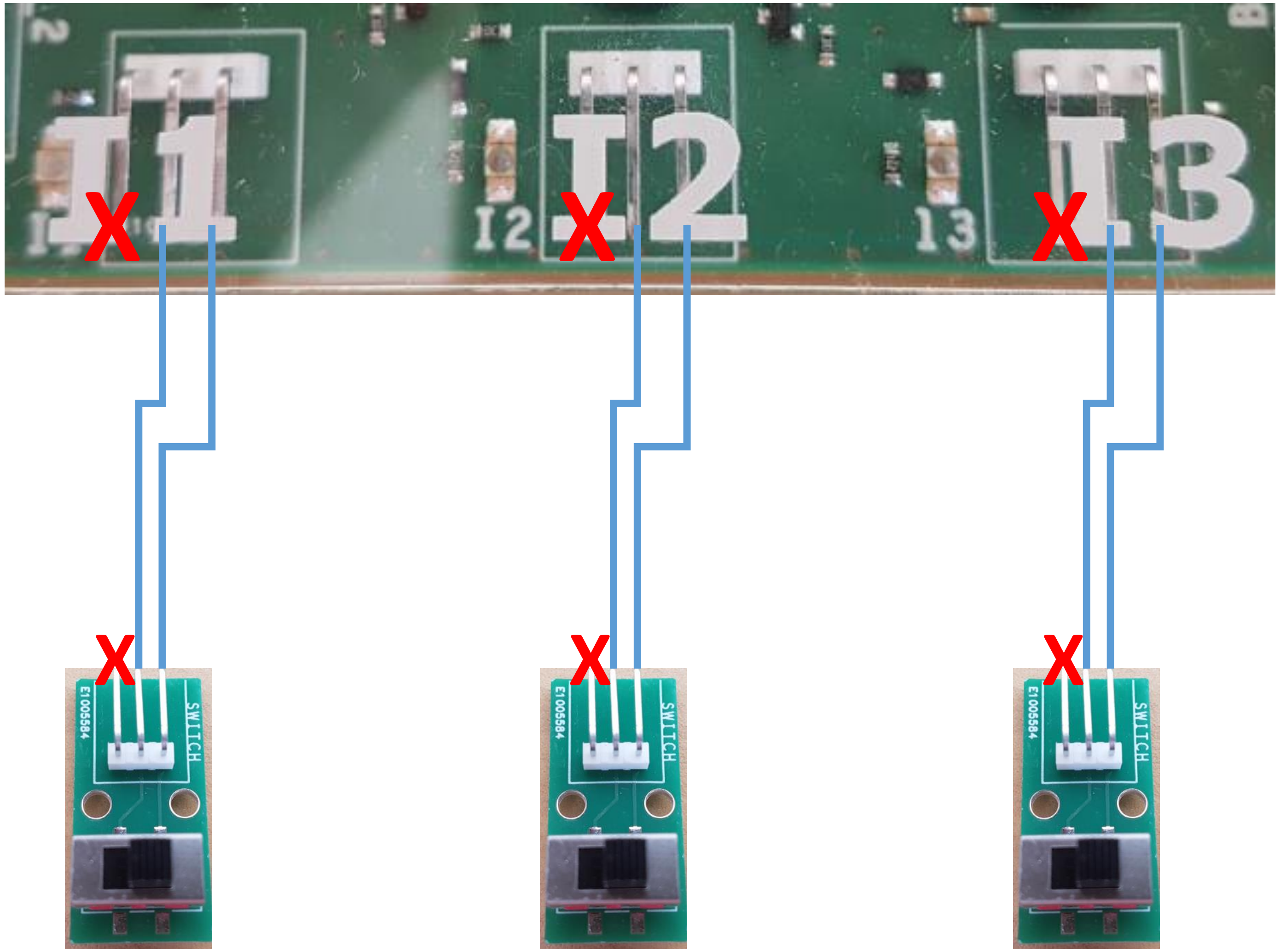
Input parameters have to be type first in the precondition clause
Syntax: **PRE** predicates **THEN** substitution **END**

Operations specified in LOCAL_OPERATIONS have to be implemented in OPERATIONS

```
OPERATIONS
    res <-- triAND(v1, v2, v3) = /* AND over 3 values */
    BEGIN
        res :( res : uint8_t);
        res := IO_OFF;
        IF v1 = IO_ON THEN
            IF v2 = IO_ON THEN
                IF v3 = IO_ON THEN
                    res := IO_ON
                END
            END
        END
    END
```

Output parameters have to be typed first

# Combinatorial example




How to connect switches to the board

# Combinatorial example

Your turn:

- **Model**:
  - Open the project combinatorial
  - Have a look at the component logic / logic_i , OPERATION user_logic
- **Prove**:
  - Ctrl+A (component view) to select all components, press F0 to start type check, PO generation and proof in sequence
- **Compile**:
  - Right click on the project (left pane) then select « Compile LCHIP M »
- **Upload**:
  - Connect your board, click on « upload », click on « connect », click on « erase program verify », reset your board, wait for « device ready », reset your board
- **Check** with your buttons that the function is correctly implemented

# Going further

Your turn:

- **Instead of a AND, program a OR over the 3 inputs**
- **Model**:
  - Rename triAND operation in triOR (LOCAL_OPERATIONS & OPERATIONS)
  - Modify triOR and user_logic implementations
- **Prove**:
  - Ctrl+A (component view) to select all components, press F0 to start type check, PO generation and proof in sequence
- **Compile**:
  - Right click on the project (left pane) then select « CSSP Runner »
- **Upload**:
  - Connect your board, click on the green arrow, reset your board, wait for « device ready », reset your board
- **Check** with your buttons that the function is correctly implemented

# Agenda

Introduction to the CLEARSY Safety Platform

Development process (demo video)
Bits of B
Using the modelling interface

The clock example (synchronous)
The combinatorial example (asynchronous)

**Conclusion**

# Conclusion

**Modelling environnement to experiment with**

- Formal methods
- Embedded systems / physical world / IoT

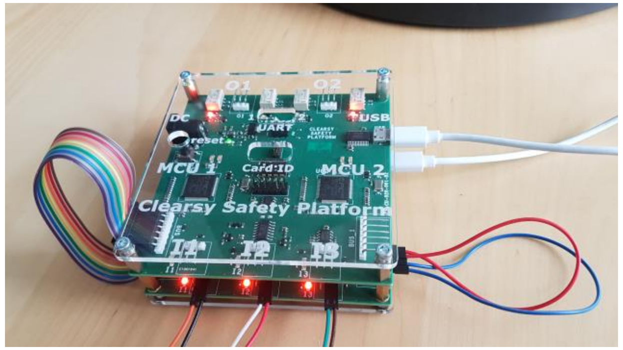
**Available as a complete IDE**

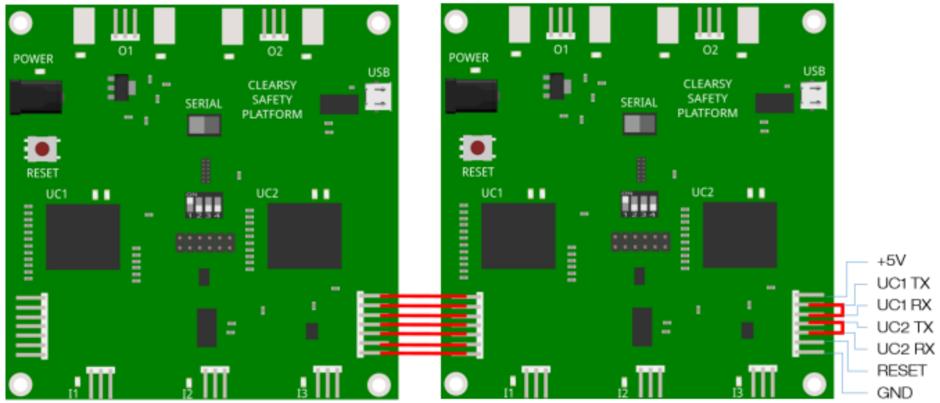
- With electronic board
- With software simulator

**More elaborated starter kit (SK2) to come in 2022:**

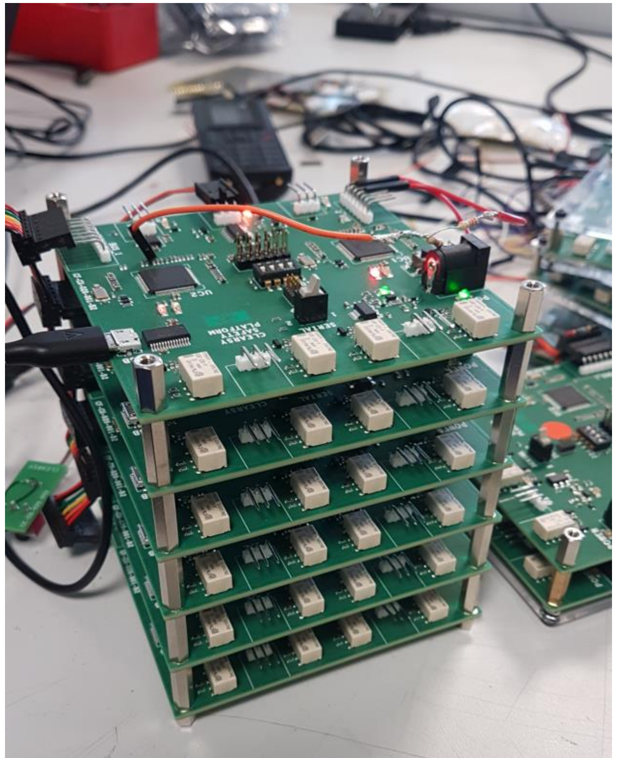
- More degrees of freedom
- Programmable in B and C
- More I/O (32 digital inputs & 32 digital outputs)

# Conclusion – further ideas



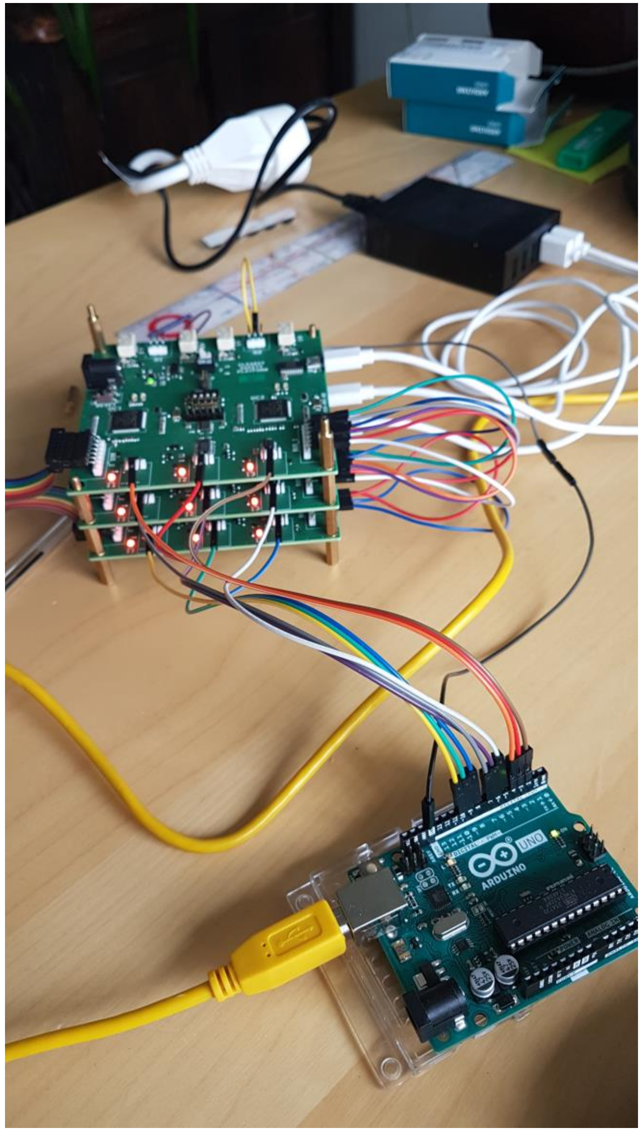

See §10. Connecting several boards together p59
https://www.clearsy.com/wp-content/uploads/2020/07/CSSP_User_Manual.pdf



## Address more I/O

- Connect several boards through their serial bus
- Configure project with multiple boards
- All boards execute the complete logic
- Inputs are distributed over several boards
- Input values are exchanged  between boards at each loop

# Conclusion – further ideas




## **Connect with other devices**

- With other SK0 boards through their I/O
- With other computers like Arduino
  - For testing or simulating environment
  - For connexion with Internet

- Examples (specification, models, schematics) to be published beginning of 2022

# CLEARSY Safety Platform For Education

*Thank you for your attention*

https://www.clearsy.com/en/our-tools/clearsy-safety-platform/

CLEARSY
SYSTEMS ENGINEERING

**Thierry Lecomte** *(thierry.lecomte@clearsy.com)*