CLEARSY
Safety Solutions Designer

Hackathon
JUL2023

AIX
LYON
PARIS
STRASBOURG

WWW.CLEARSY.COM

Tools
CSSP IDE, ProB

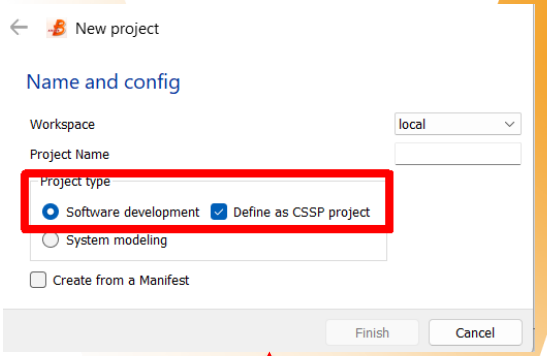
Thierry Lecomte
R&D Director

THIERRY.LECOMTE@CLEARSY.COM

Atelier CSSP IDE

≡ On a USB key (you keep it)

- Go to E:\CSSP_for_education_20230522\CSSP
- Execute **Register_CSSP.cmd**
- Execute **startAB.cmd**
- If asked, relocate the projects directory to `E:\CSSP_for_education_20230522\CSSP\CSSP_WORKSPACE`
- Open Atelier B / preferences / project, change Default Project Directory with `E:\CSSP_for_education_20230522\CSSP\CSSP_WORKSPACE`
- The CSSP projects need to be all located in this directory
- Create a project, select project type “SW development” and “define as CSSP project”



Atelier CSSP IDE

- Atelier B with abilities to deal with CLEARSY Safety Platform (CSSP)
  - Specific pre-filled projects
  - Dedicated compilation toolchain and man-machine interface
  - Programming the CLEARSY Safety Platform
  - Graphical simulation (only one microcontroller): SK0 emulation

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 4.6 – 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) ≠ $CRC_{expected}$ (code) (deferred action)
- Etc.

Handle failures:

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

Hyp

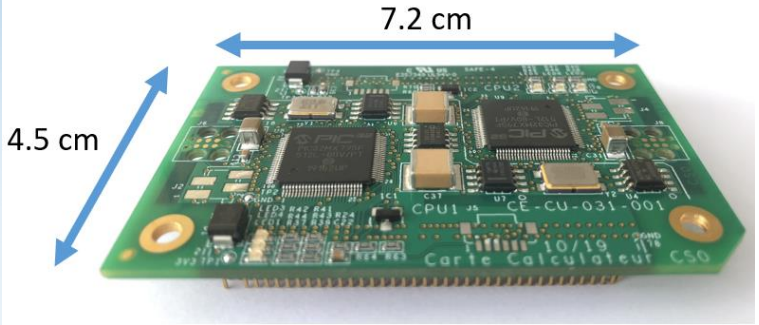
Models are proved to be correct:

- 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

SK0 board

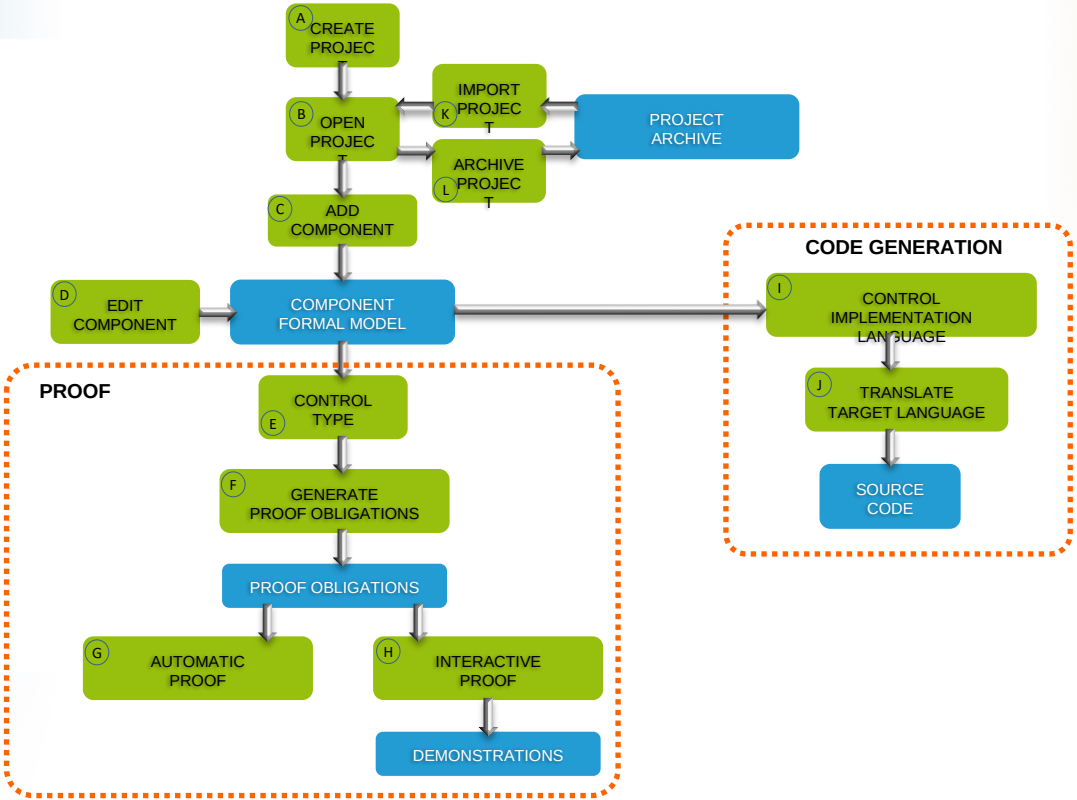


CS0 core computer

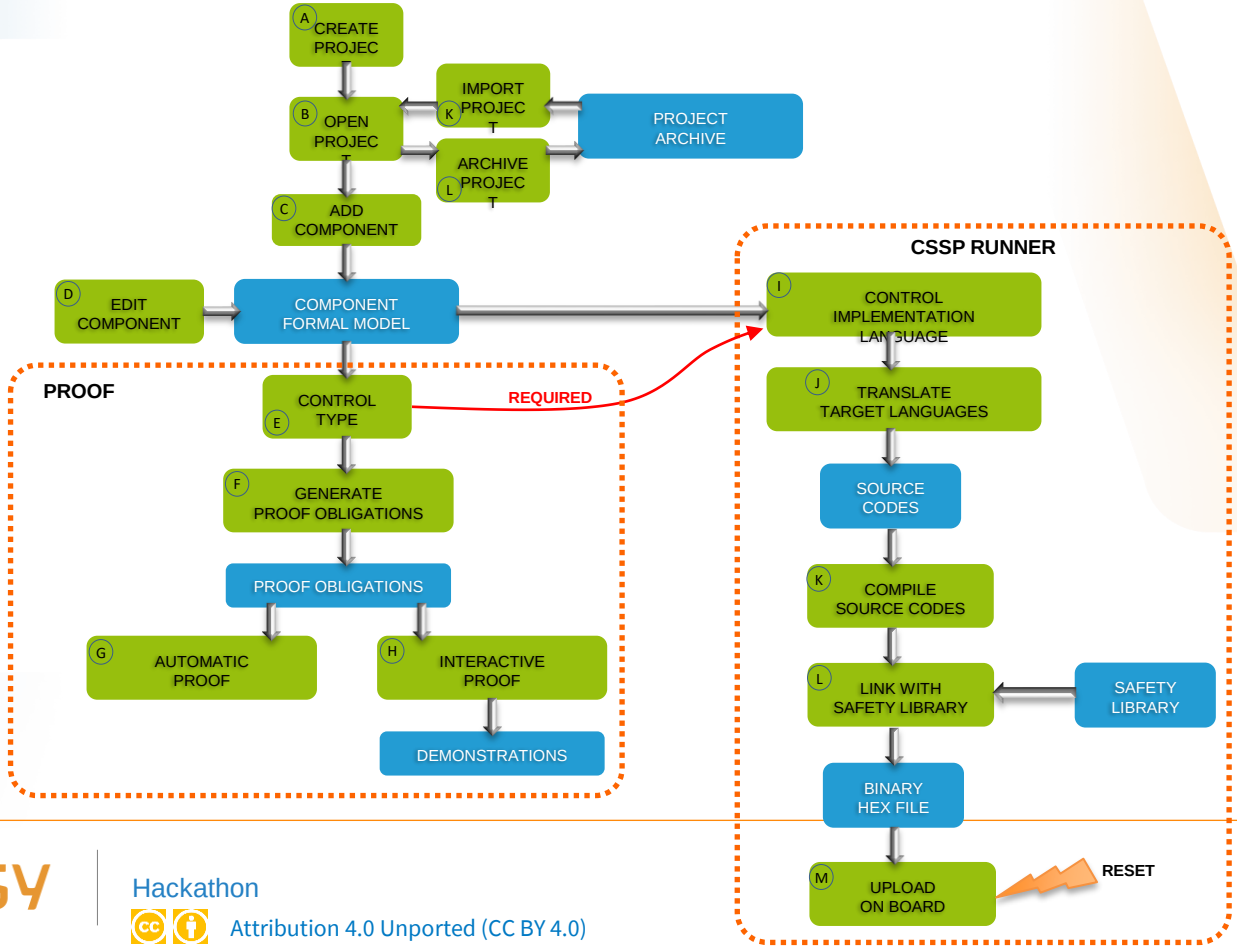
https://www.clearsy.com/en/our-tools/clearsy-safety-platform/

- **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

Development Process (B)



Development Process (B+CSSP)

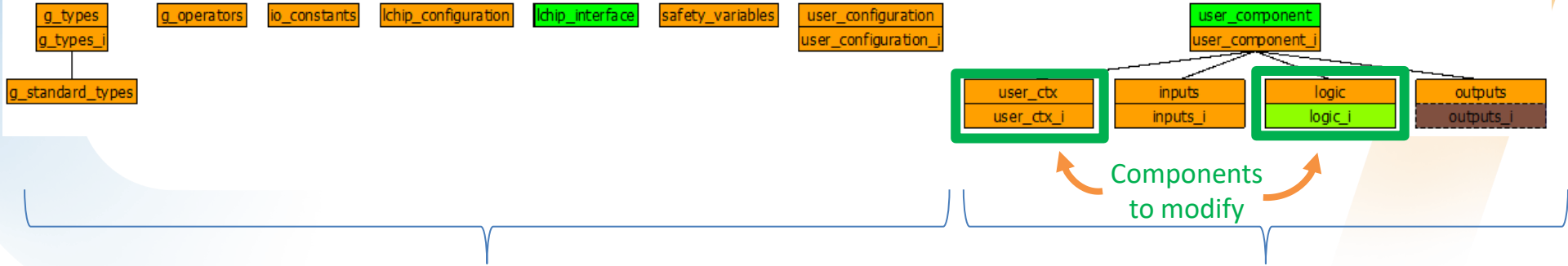


CSSP Project

A CSSP project is a B project

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

It contains



Programming Model & Applications

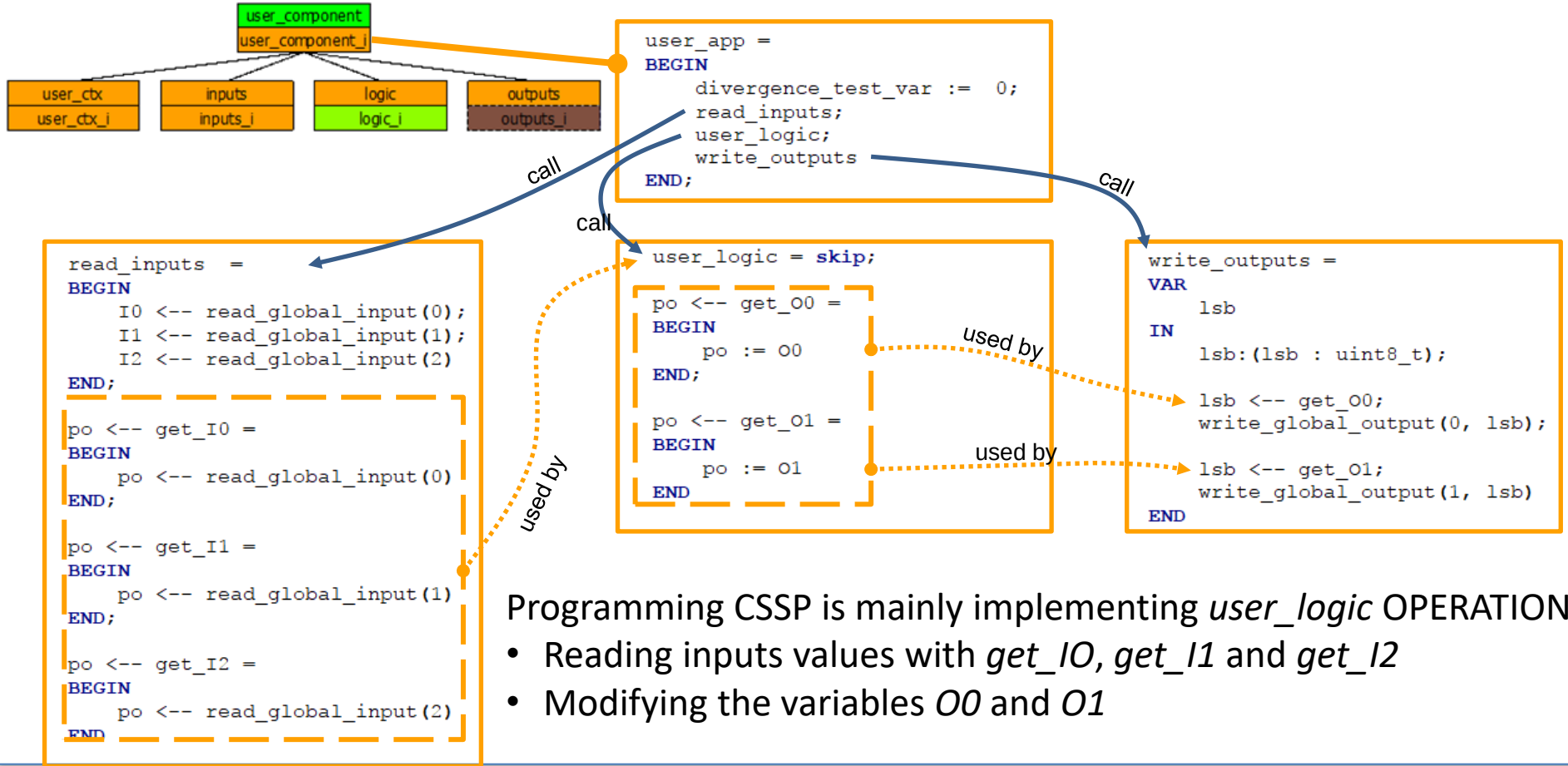
- The execution is cyclic
- The function is executed regularly as often as possible similar to arduino programming (setup(), loop())

```
init();

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

- 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*

Generated Models



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*

B Variables Declaration

specification

```
ABSTRACT_VARIABLES
    O0,
    O1

INVARIANT
    O0 : uint8_t &
    O1 : uint8_t

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

: means « belongs to »

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

:: means « any value within »

implementation

```
// pragma SAFETY_VARS

CONCRETE_VARIABLES
    O0,
    O1,
    TIME_A,
    STATUS

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
```

Mandatory
Contains variables that will be verified

Variables local to implementation

B Constants Declaration

specification

```
CONCRETE_CONSTANTS
    DELTA_T

PROPERTIES
    DELTA_T : uint32_t
```

implementation

```
// pragma CONSTANTS
```

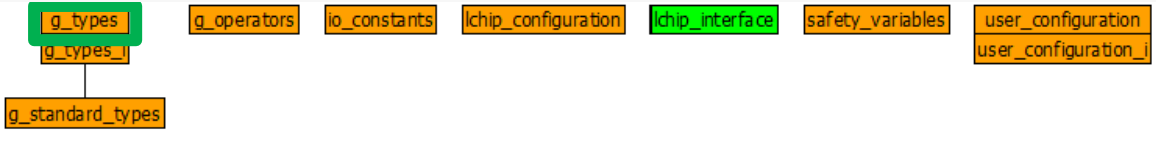
Mandatory
Contains constants that will be verified

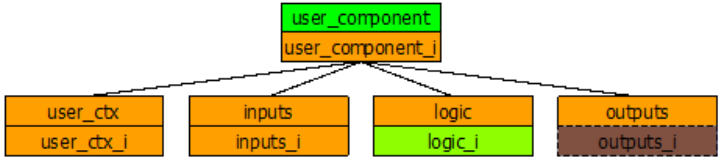
Important
A model cannot contain both variables and constants

```
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
```

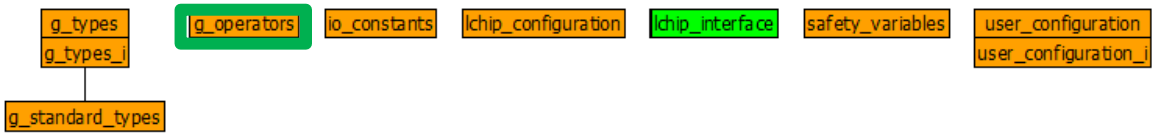
Everything is either 8, 16 or 32 bits

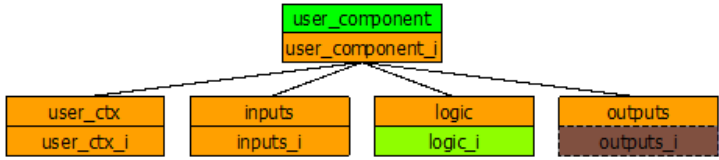
Boolean values TRUE and FALSE coded on 8 bits

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 &
```

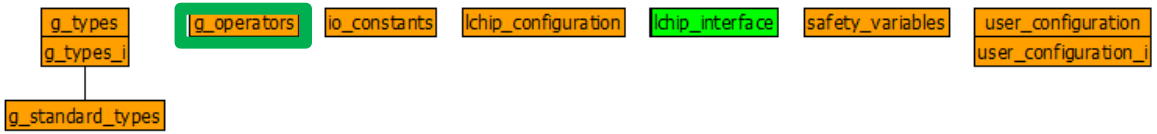
```
bitwise_not_uint32 : uint32_t --> uint32_t
```

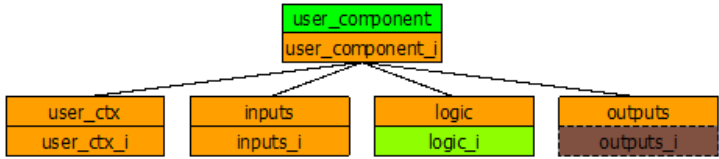
total function that associates a 32-bit unsigned integer to any 32-bit unsigned integer

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

total function with two 32-bit unsigned integer parameters

Unsigned INT Operators





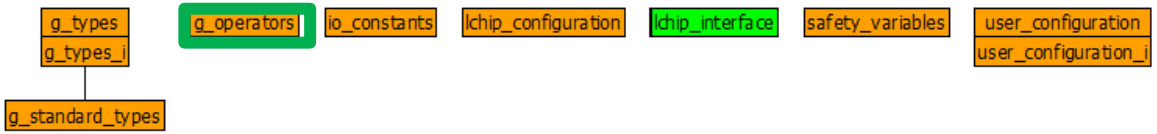

```
CONCRETE_CONSTANTS
    bitwise_not_uint32,
    bitwise_and_uint32,
    bitwise_xor_uint32,
    bitwise_not_uint16,
    bitwise_and_uint16,
    bitwise_xor_uint16,
    bitwise_or_uint16,
    bitwise_not_uint8,
    bitwise_and_uint8,
    bitwise_xor_uint8,
    bitwise_or_uint8,
    add_uint32,
   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

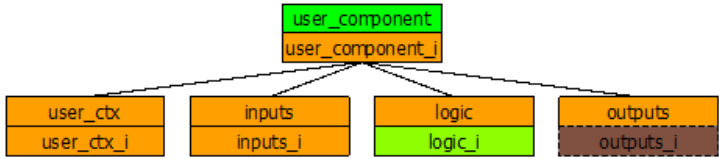
Builtin operators

```
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)) &
```

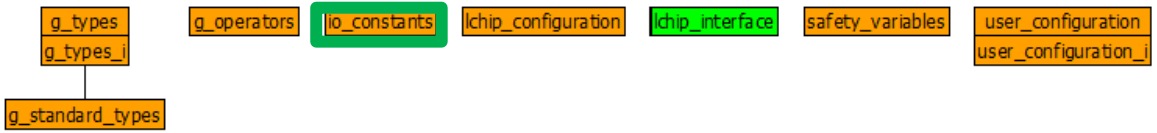
```
add_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 + x2) mod (MAX_UINT32 + 1))
```

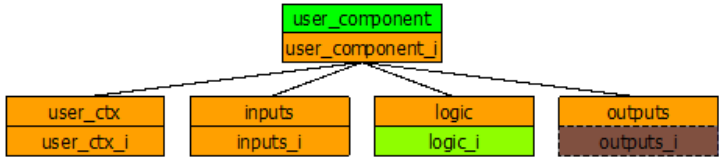
is a λ function

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

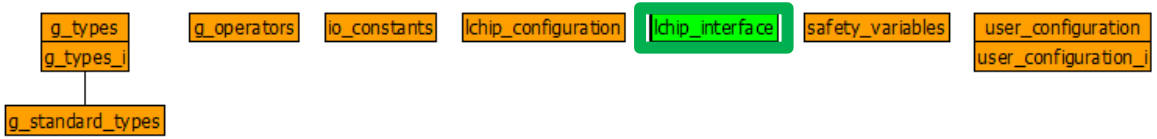
values used by digital inputs and outputs

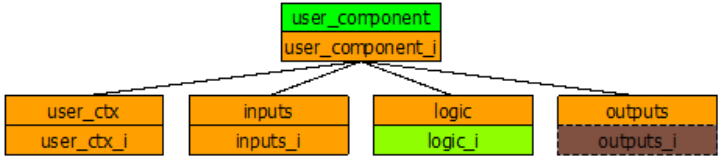
coded on 8 bits

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

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

:( ) means « becomes such that »

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

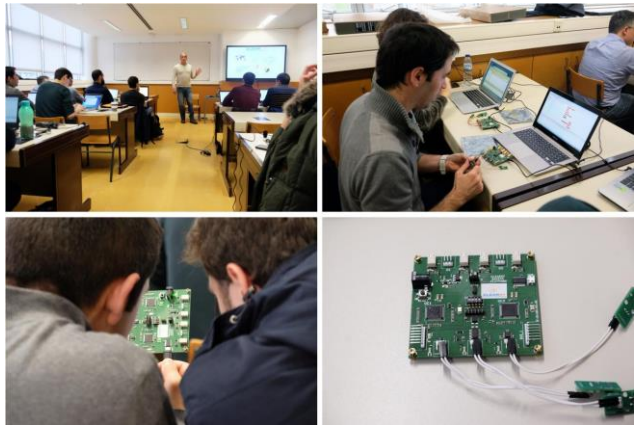
```
user_logic =
BEGIN
    O0 := IO_ON;
    O1 := IO_OFF
END
```

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

Links and References

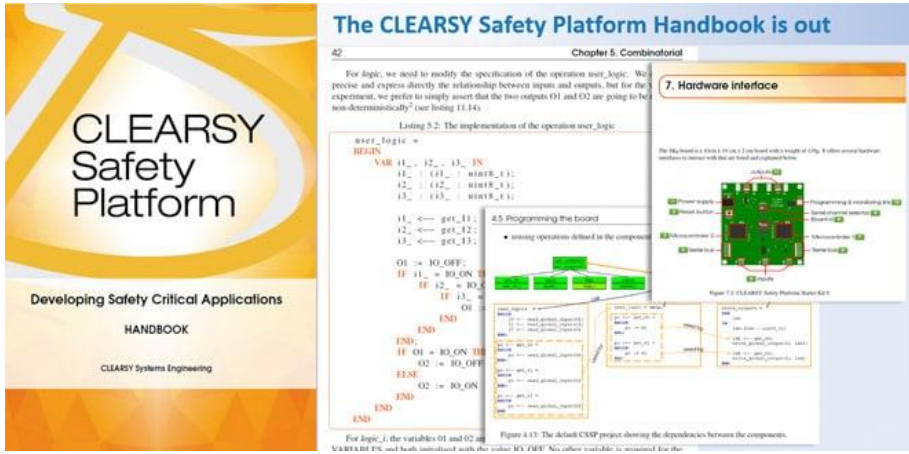
► On going courses / research



HASLab, University of Minho
Braga, Portugal, March 2019

INESCTEC

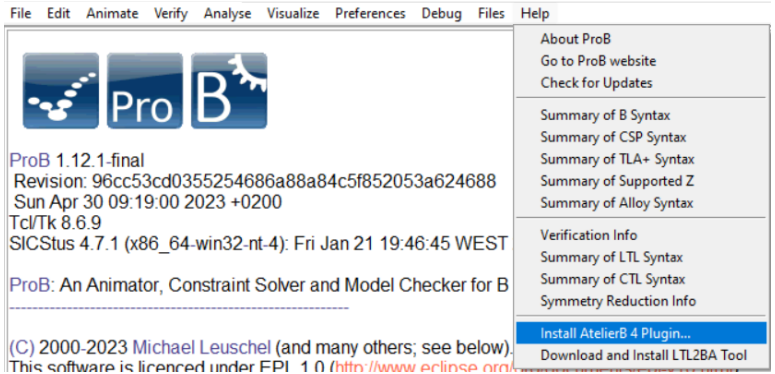
► Available resources

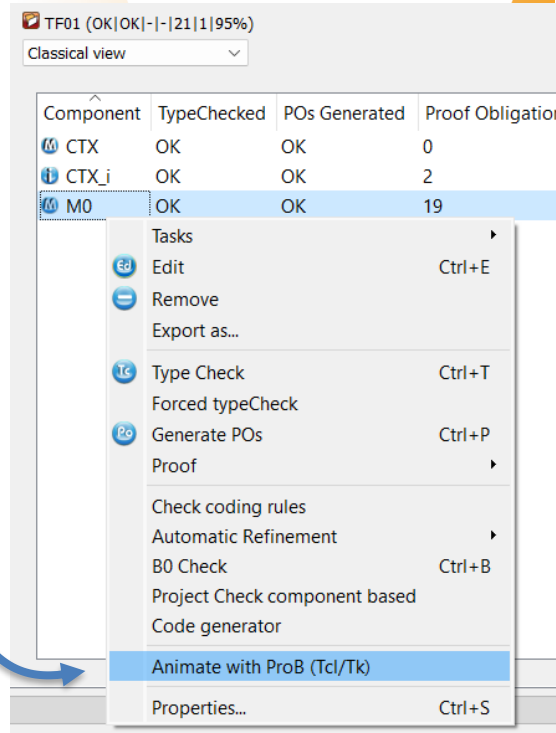


https://github.com/CLEARSY/CSSP-Programming-Handbook

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

Add Prob Tool to Atelier B menu





- Open ProB Help menu
- Install Atelier B 4 plugin
- Select the Atelier B file of your install directory
  E:\CSSP_for_education_20230522\CSSP\Atelier_B_cssp_4.6.0-rc7\AtelierB
- probtclk.etool file created in extensions subdirectory