CLEARSY
Safety Solutions Designer

TUTORIAL

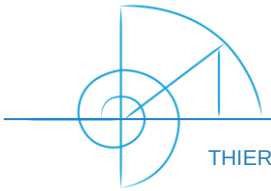
ABZ 2021

JUNE 2021

AIX
LYON
PARIS
STRASBOURG

WWW.CLEARSY.COM

Using B to program the CLEARSY Safety Platform



THIERRY.LECOMTE@CLEARSY.COM

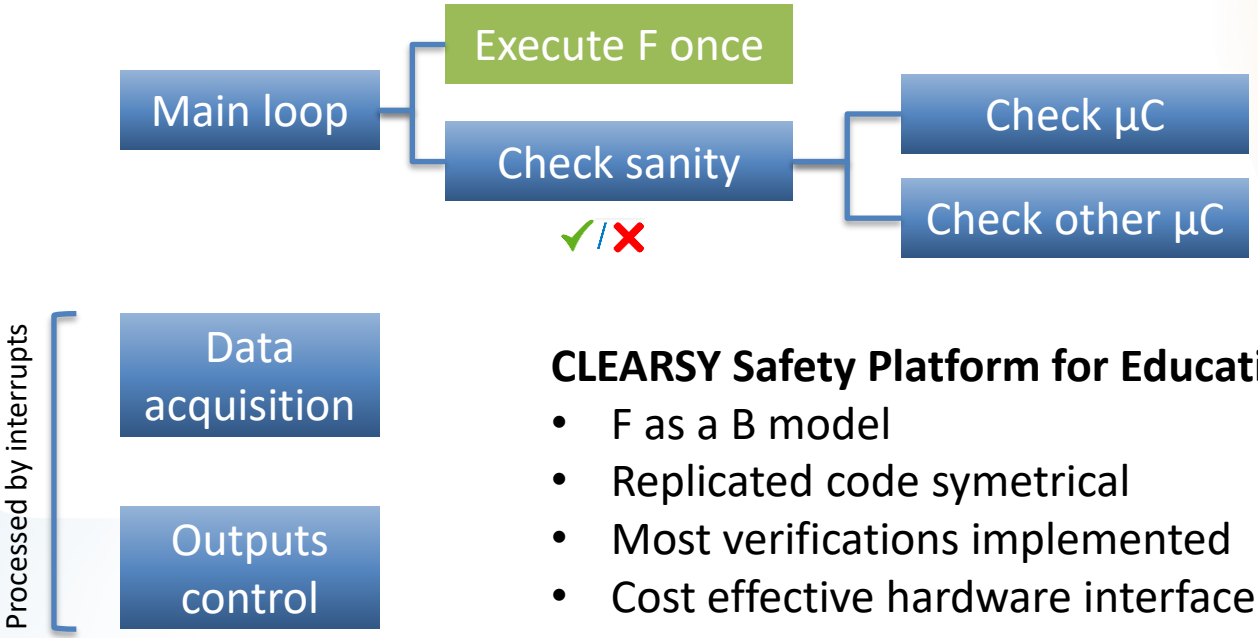
PART II

THE CLEARSY SAFETY PLAFORM FOR EDUCATION

Main Principles: software

Legend

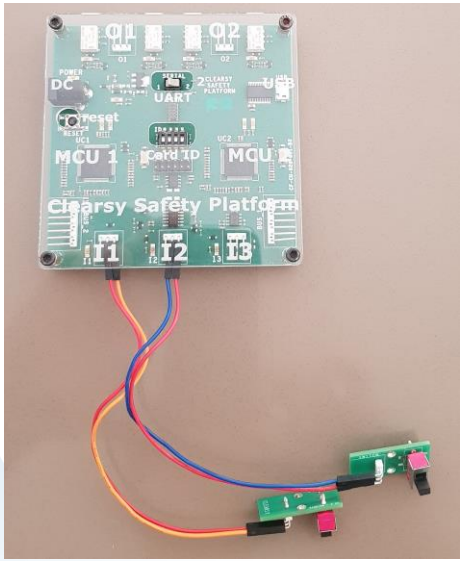
- to develop
- to complement
- developped or generated



CLEARSY Safety Platform for Education

- F as a B model
- Replicated code symetrical
- Most verifications implemented
- Cost effective hardware interface

Required Components

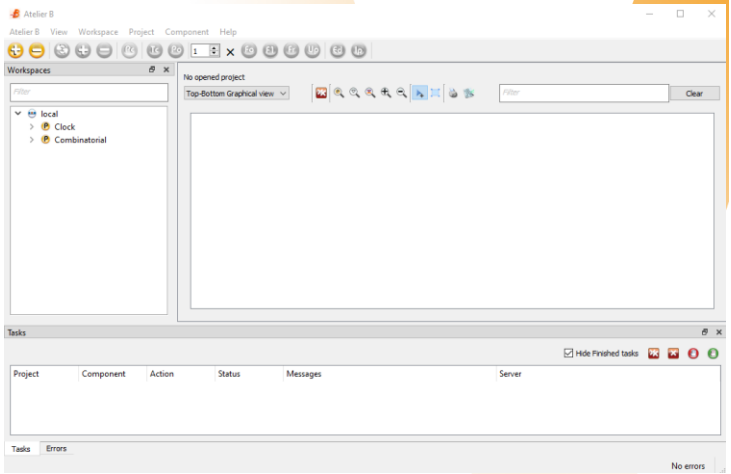


- 1x SK0 board (optional)
- 1x USB cable
- 1x laptop
- 2 switches
- IDE

https://github.com/CLEARSY/tutorial-ABZ-2021
Section « Atelier CLEARSY Safety Platform »

Atelier CLEARSY Safety Platform

- Zip file + install procedure
  - Unzip in a directory without space nor special chars in the path
  - Run Register_CSSP.cmd script
  - Run startAB.cmd script just created
- CSSP skeleton project
  - Create a project
  - Modify 2 components
  - Prove, compile
  - Upload and execute on the SK0 board
  - Execute with SK0 software emulator (1 replica)



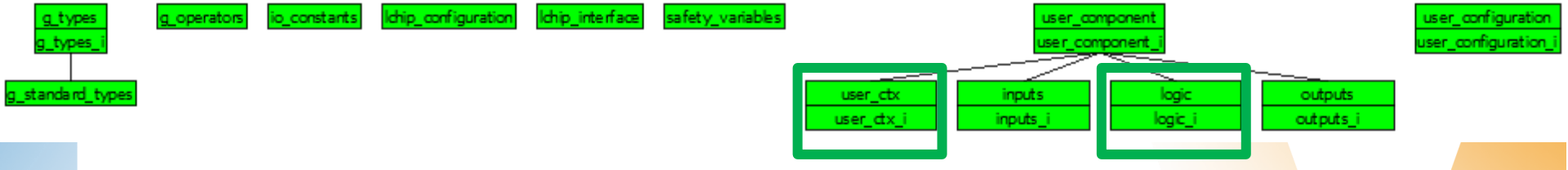
IDE populated with 2 examples

Development cycle

- Create a CSSP software project
- Configure the board
  - I/O used, naming
- Generate the skeleton project
- Define the constants in user_ctx component* (nothing to do here)
- Define the behaviour in the logic component
- Prove, compile, upload, execute

(*) : in B, a component is a specification and its implementation

Project architecture



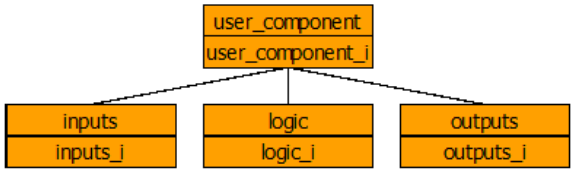
Components to modify

Interface with the safety library

Transfer Function F

Function model

Syntax:
pp <-- ff(vv)
represents a call to operation ff(vv) that returns the value pp




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

call

call

call

```
read_inputs =
BEGIN
    AN <-- read_global_input(0);
    FRE <-- read_global_input(1)
END;

po <-- get_AN =
BEGIN
    po := AN
END;

po <-- get_FRE =
BEGIN
    po := FRE
END
```

used by

```
user_logic = skip;
po <-- get_ENG =
BEGIN
    po := ENG
END;

po <-- get_SIG =
BEGIN
    po := SIG
END
```

used by

used by

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

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

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

Some modelling elements

- Typing
  - uint8_t, uint16_t, uint32_t
  - I/O are uint8_t with IO_OFF and IO_ON valid states (using other values leads to panic mode during execution)
- Predefined arithmetic and bitwise operations
  - add_uint8 computes addition of 2x uint8_t without overflow
- Local variables have to be typed with predicate

```
write_outputs =
VAR
    lsb
IN
    lsb :( lsb : uint8_t );
    lsb <-- get_ENG;
```

Introduction to 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

Introduction to B: operation specification

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 when the operation is completed 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)

Introduction to B: operation implementation

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

```
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

Only single condition (no conjonction nor disjonction)

= < <= operators only

Programming the board

Video « Programming SK0»

The behavioural model

```
OPERATIONS
    user_logic =
    BEGIN
        ENG, SIG, COUNTER :(
            ENG : uint8_t &
            SIG : uint8_t &
            COUNTER : uint8_t &
            ((ENG = IO_OFF & SIG = IO_ON) or (ENG = IO_ON & SIG = IO_OFF)) &
            (COUNTER = 0 <=> (ENG = IO_ON & SIG = IO_OFF))
        )
    END;
```

Component Status for logic

AutoProved C:/Users/thier/AB_CSSP_4.6.0-RC7/XING_SK0/logic.mch

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
logic	**1**	**0**	**1**	**0**	**100**

```
OPERATIONS
    user_logic =
    VAR an_, fre_ IN
        an_ :(an_ : uint8_t);
        fre_ :(fre_ : uint8_t);
        an_ <-- get_AN;
        fre_ <-- get_FRE;

        IF prevAN = IO_OFF THEN
            IF an_ = IO_ON THEN
15/15           COUNTER := add_uint8(COUNTER , 1)
            END
        END;

        IF prevFRE = IO_OFF THEN
            IF fre_ = IO_ON THEN
18/18           COUNTER := sub_uint8(COUNTER , 1)
            END
        END;

        prevFRE := fre_;
        prevAN := an_;

        IF COUNTER = 0 THEN
            ENG := IO_ON;
            SIG := IO_OFF
        ELSE
 9/9        ENG := IO_OFF;
 9/9        SIG := IO_ON
        END
    END;
```

Component Status for logic_i

AutoProved C:/Users/thier/AB_CSSP_4.6.0-RC7/XING_SK0/logic_i.imp

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
Operation_user_logic	29	0	29	0	100
WellDefinedness_user_logic	8	0	8	0	100
logic_i	**38**	**0**	**38**	**0**	**100**

Project on github to restore IDE (src/XING_SK0.arc)

Use menu *workspace/restore project*

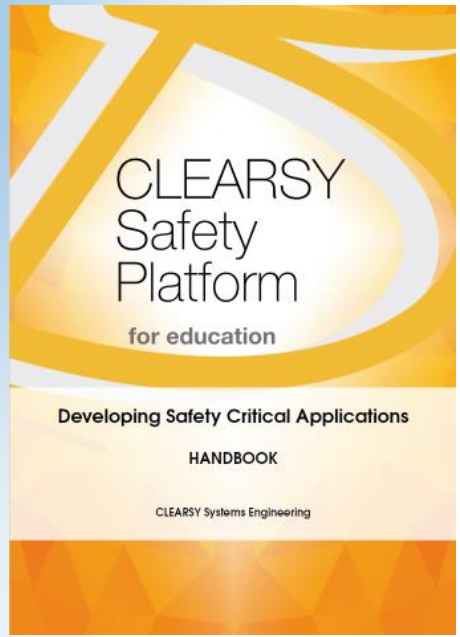
Falling back to restrictive mode

- Are safety mechanisms really working ?
- Triggers:
  - Very long computation (iterate through a table)
  - Use an invalid value for output (different from IO_OFF, IO_ON)
  - Use built-in functions *get_replica_id* and *get_processor_id* to program different behaviours on different replicas
- Only works with the SK0 board
  - The software emulator only executes one replica

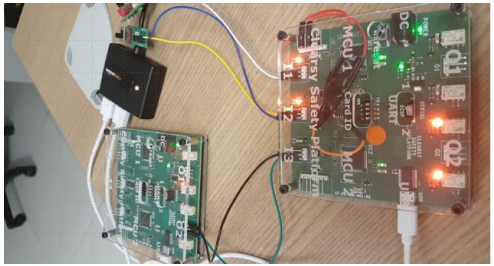
Programming the board

Video « Triggering panic mode»

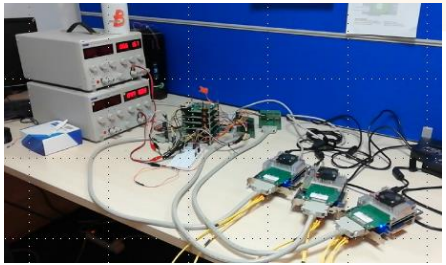
Going further



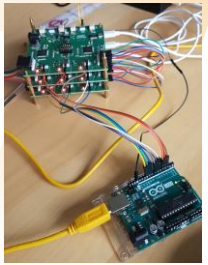
- Programming model
- Software and hardware interfaces
- Examples
- https://github.com/CLEARSY/CSSP-Programming-Handbook



Used in combination



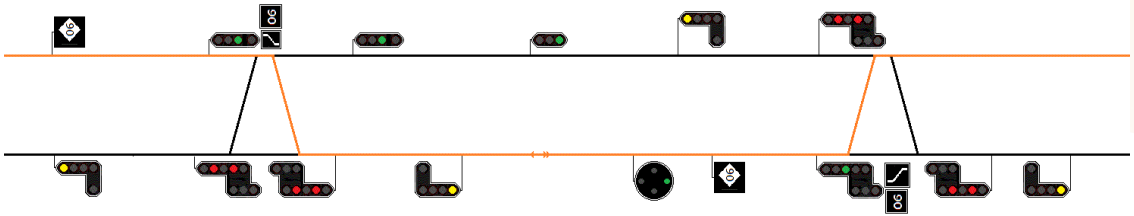
Remote I/O system



Stimulated with arduino

Fit for Proof of Concept

- B0 as Internal Representation
  - B0 could be generated / translated from other formalisms
- PoC for SNCF
  - Relay-based installation for temporary works



  - Printed drawings as inputs
  - Some manual interaction, translation, execution of the wired logic

Programming the board

Video « supporting legacy signalling drawings»