RSSR 2017

LCHIP

LOW-COST HIGH-INTEGRITY PLATFORM

Tutorial #2

Pistoia, November 14, 2017

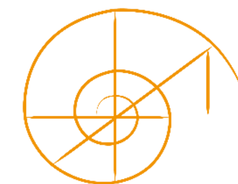
Thierry Lecomte & Patrick péronne, ClearSy
{thierry.lecomte, patrick.peronne}@clearsy.com

LCHIP is the name of a French R&D project (FUI21) funded by Bpifrance, the public bank for innovation

Funded by

Région Provence Alpes Côte d'Azur

MÉTROPOLE AIX-MARSEILLE PROVENCE



CLEARSY
SYSTEMS ENGINEERING

Agenda

Thierry Lecomte
R&D Director
ClearSy

Patrick Péronne
Safety Engineer
ClearSy

LCHIP
Starter kit SK_0

≡ Introduction to LCHIP

- Genesis
- LCHIP in a nutshell
- Roadmap
- Starter Kit SK0
- A complete transaction (LCHIP in action)

≡ Programming

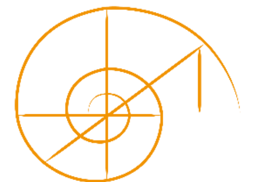
- The programming model
- A LCHIP project
- Function model
- Introduction to B: variables, constants, types, uint operators, inputs, outputs, operations, user_logic
- Troubleshooting

≡ Use case 1: Combinatorial

≡ Use case 2: Clock

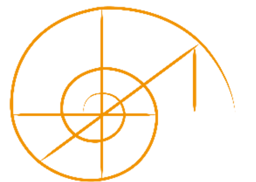
≡ Use case 3: …

Disclaimer
Even if the added value of LCHIP is due to the mathematical proof of the software, in this tutorial we are mainly aimed at the Toolchain generating the application.
The tutorial for the next version version of the starter kit will be more proof oriented



CLEARSY
SYSTEMS ENGINEERING

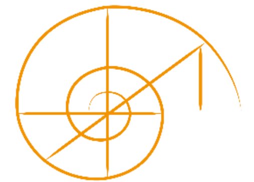
Introduction to LCHIP



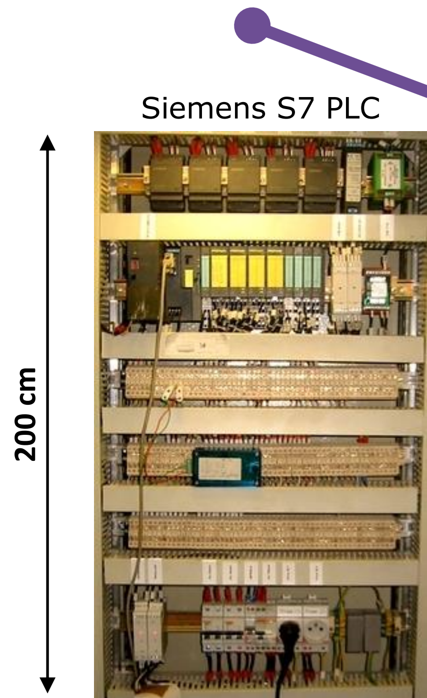
Genesis

- Need for a technical solution to overcome difficulties to develop SIL3/SIL4 systems
  - Require rare human resources to complete successfully
  - Very short delays (~6 months) to design new systems
  - Off-the-shelf block solutions difficult to adapt

Technology based on double-processor and formal method

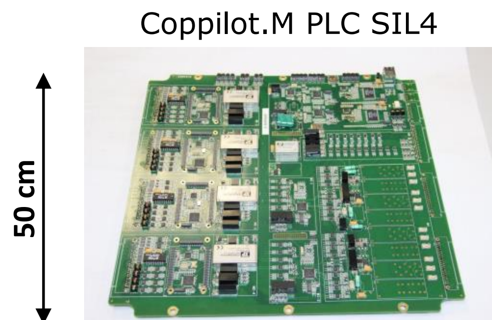


Siemens S7 PLC



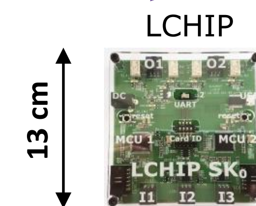
Sao Paulo L2 & L3 (2009)

Coppilot.M PLC SIL4



Sao Paulo L15 (2016)
Stockholm Citybanan (2017)

LCHIP



2017

Genesis

Technology experimented several times

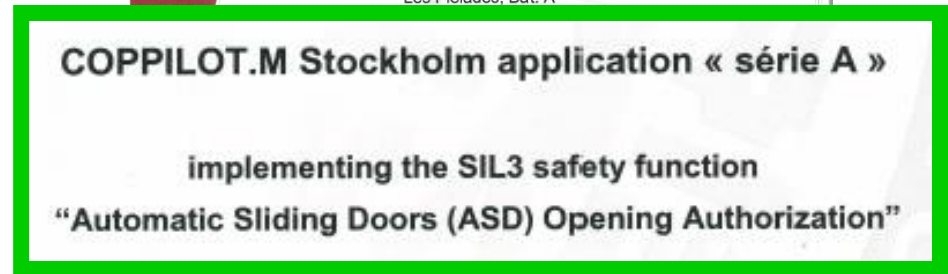
Platform screen doors controler installed in Stockholm (Citybanan)

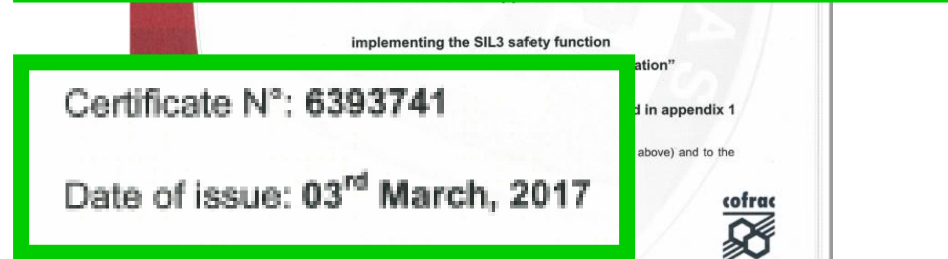
Sao Paulo L15 (2016)

Stockholm Citybanan (2017)

Gateway SATURN (2016)

Technology certifiable

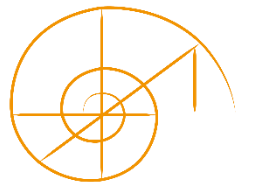




Genesis

Next Step

Develop a generic version of this technology
that could fit a broader range of applications



CLEARSY
SYSTEMS ENGINEERING

LCHIP in a Nutshell

SOFTWARE & HARDWARE
PLATFORM
FOR SAFETY CRITICAL APPLICATIONS

Factory to generate a function that executes safely

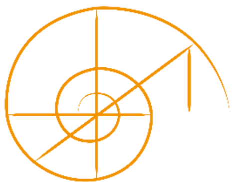
DESIGN ► ***EXECUTION***

≡ ease the development of safety critical applications

- Cover the whole development cycle
- Safety principles are built-in
- Based on a formal language and related proof tools
- Mathematical proof replaces unit and integration testing
- Avoid expert transactions over multiple languages

≡ ease the certification of safety critical applications

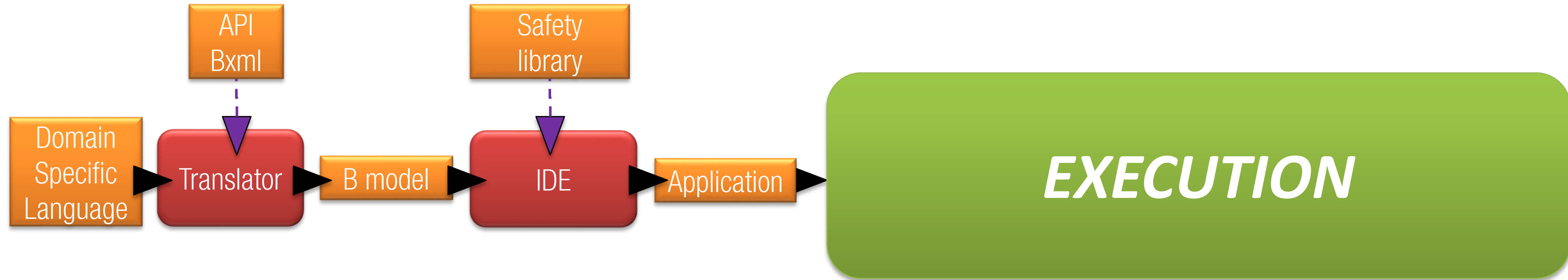
- Safety cannot be altered by the developer
- Comes along with a certification kit



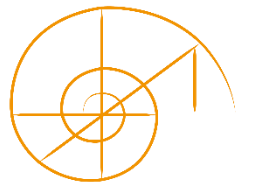
Disclaimer: LCHIP is not SIL3/SIL4 by itself. The developer is responsible for the safety demonstration

CLEARSY
SYSTEMS ENGINEERING

LCHIP in a Nutshell



- Developer focus on the function to develop by using its usual tools / language
- Transformation of the DSL inputs into B formal models, proof, code generation and compilation are fully automatic
  - Some constraints to comply though



LCHIP in a Nutshell

Design

Execution

Domain Specific Language → Translator → B model → IDE → Application → Bootloader

API Bxml

Safety library

LCHIP board

µC1

Programme

Binaire 1

Séquenceur + Librairie sécurité

Binaire 2

µC2

Programme

Binaire 1

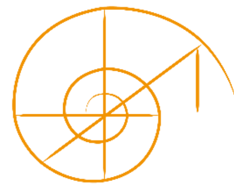
Séquenceur + Librairie sécurité

Binaire 2

Inputs
Outputs

Maintenance processor

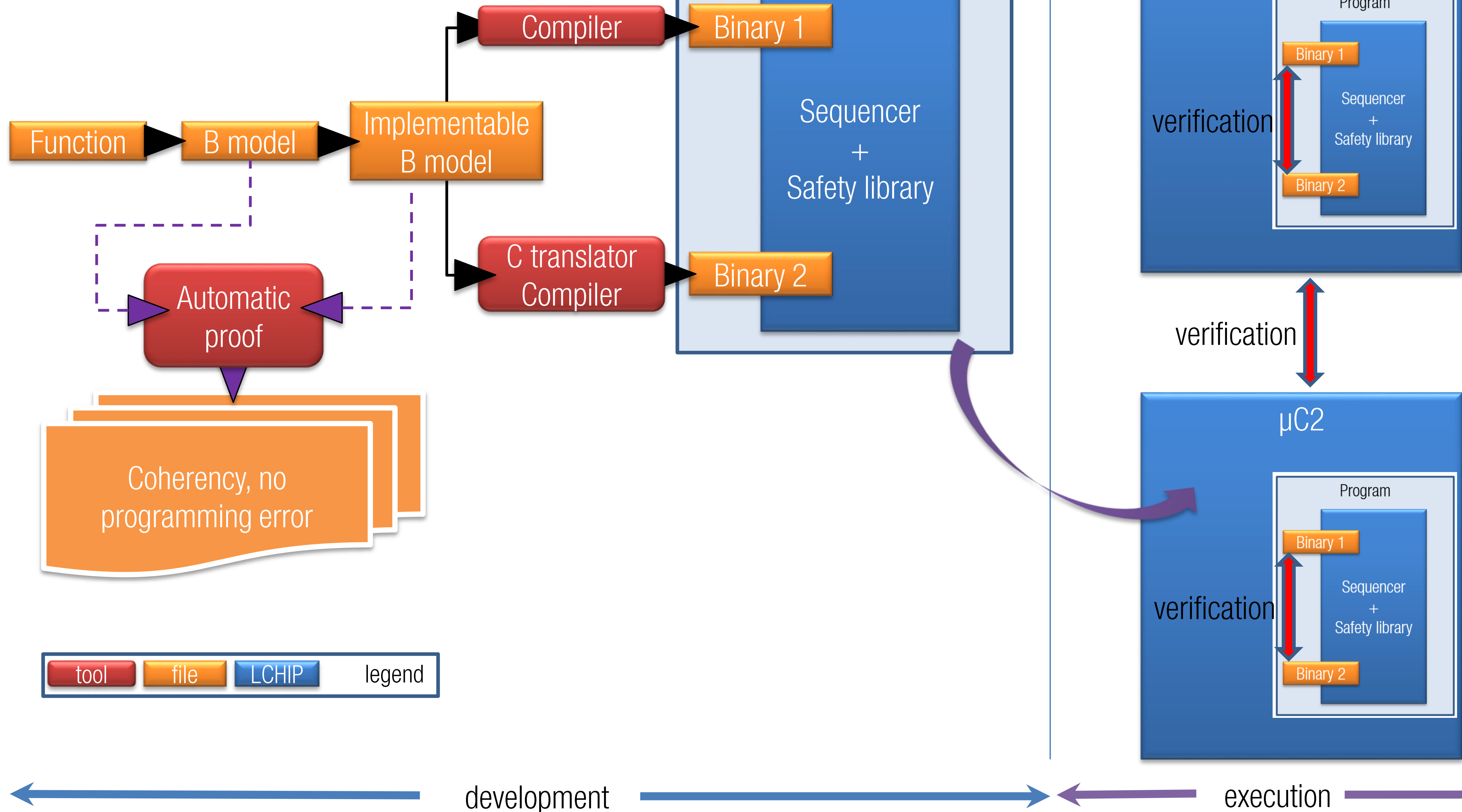
- The function is executed on the LCHIP board on 2x PIC 32 microcontrollers
  - 100 MIPS available
- Divergent or inconsistent behaviour lead to reboot
- Remote means to log and analyse the LCHIP board



Supervision
SCADA

Technical principles

To beam up a function on 2 microcontrolers



The Software Factory

Application

End User

Natural language specification

Abstract Formal model

proof

B

Refining tool

Concrete Formal model

proof

B

Code generator

Compiler

Source code

C

Binary code

HEX

Binary code

ASM MIPS, HEX

Compiler

Linker

Application

HEX

Safety Belt

IDE dev

Abstract Formal model

proof

B

Concrete Formal model

proof

B

Code generator

Compiler

Source code

C

Binary code

HEX

Source code

C

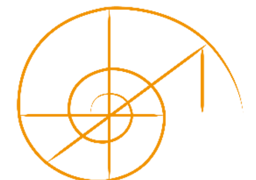
Binary code

HEX

IDE dev

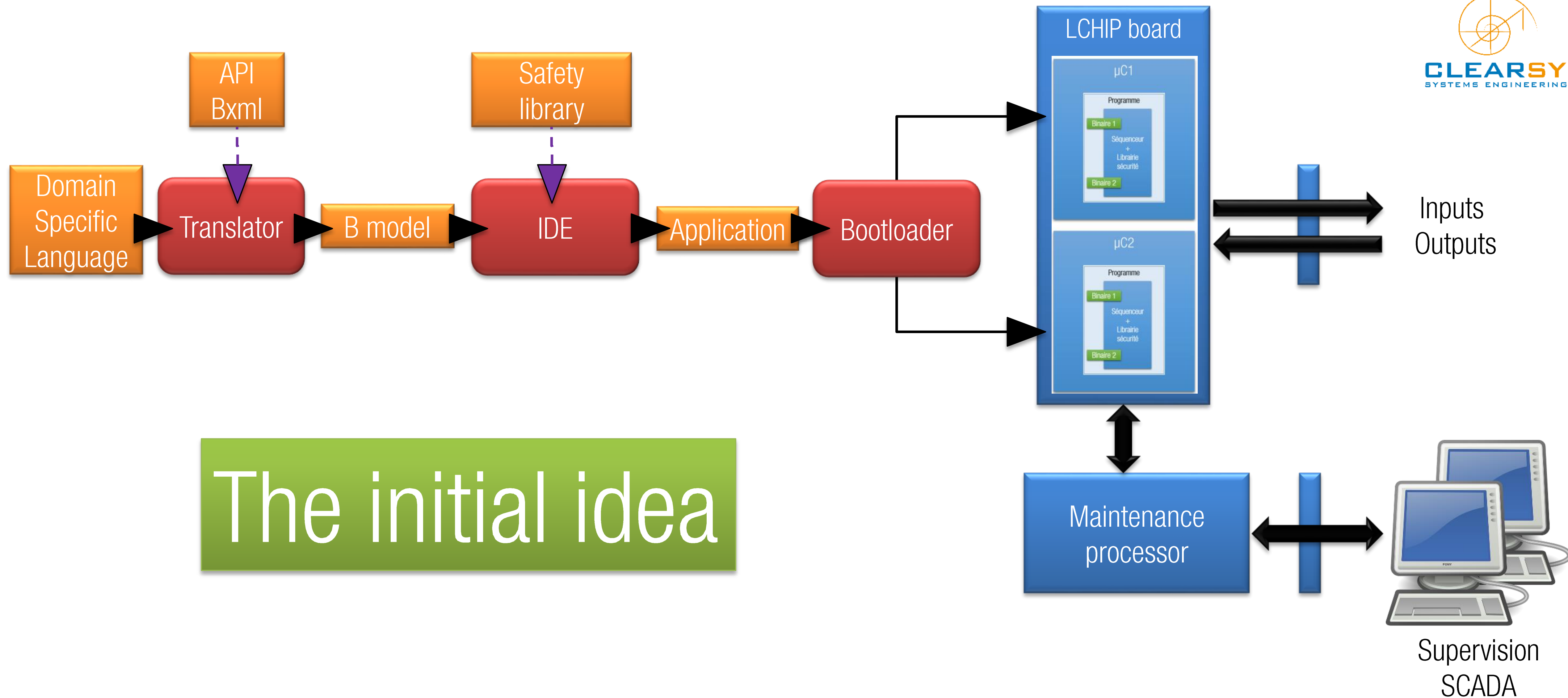
LIBRARY

Memory mapping
Cross verification
CRC



CLEARSY
SYSTEMS ENGINEERING

Roadmap



Roadmap

Safety library

B model → IDE → Application → Bootloader

Proof partly automatic

LCHIP board

µC1

Programme

Binaire 1

Séquenceur + Librairie sécurité

Binaire 2

µC2

Programme

Binaire 1

Séquenceur + Librairie sécurité

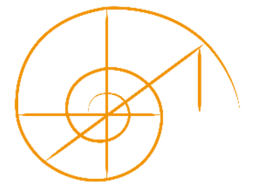
Binaire 2

Inputs (3x)

Outputs (2x)

Digital I/O

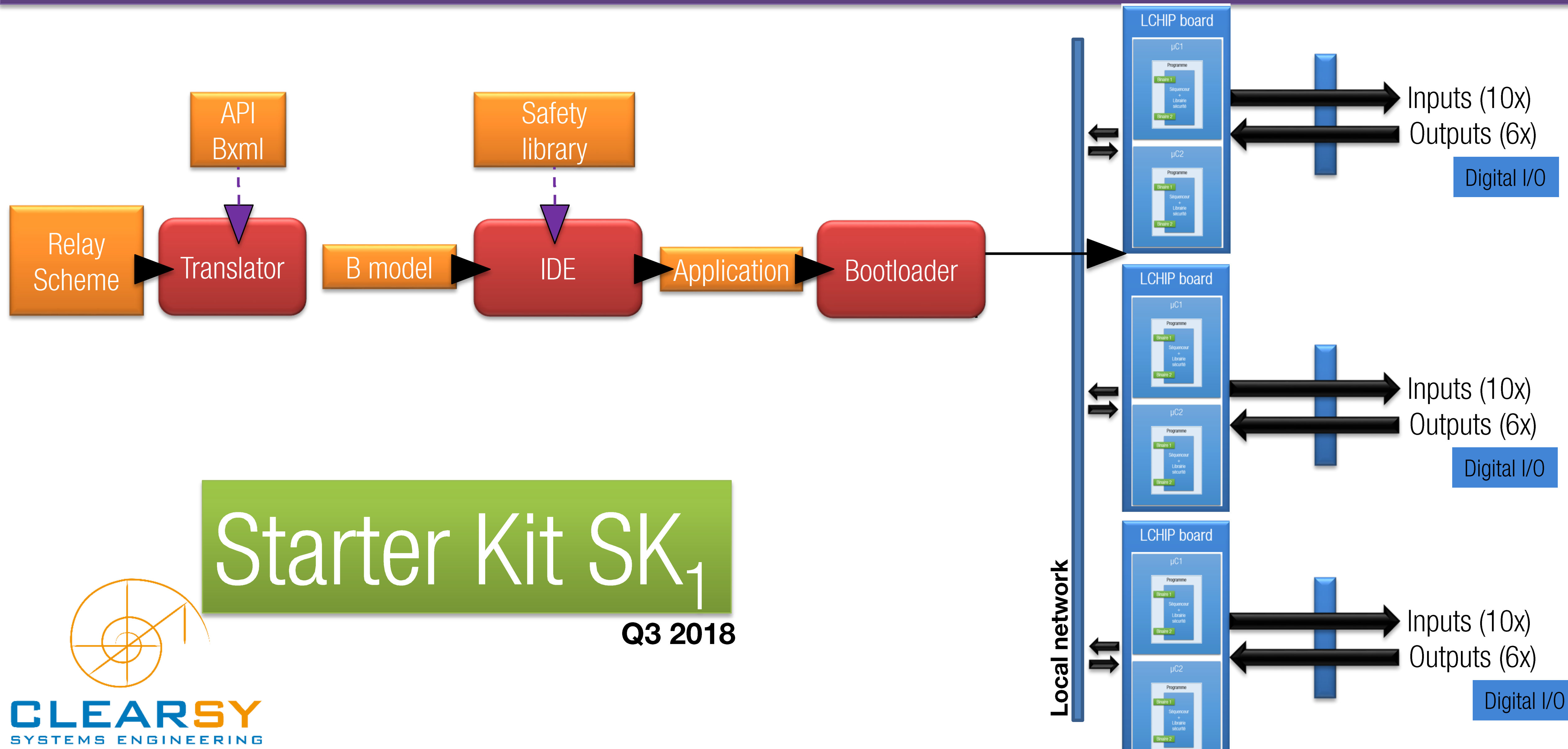
Starter Kit SK_0



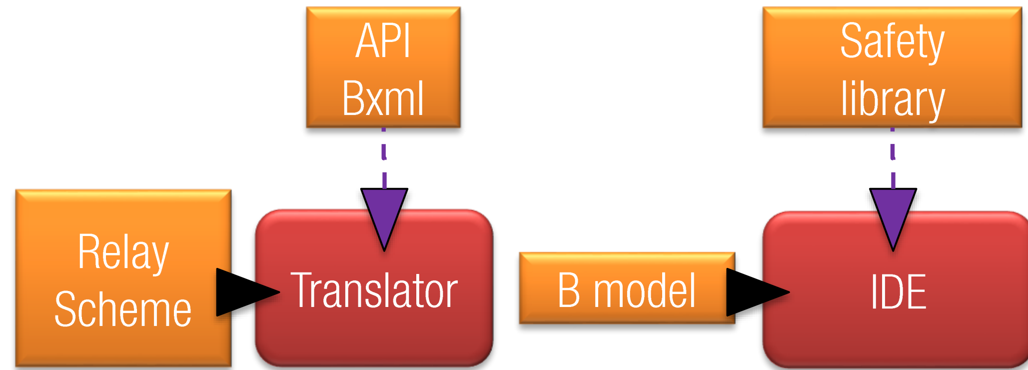
CLEARSY

SYSTEMS ENGINEERING

Roadmap

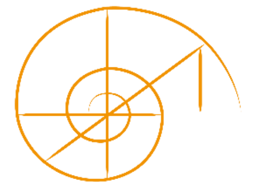


Roadmap

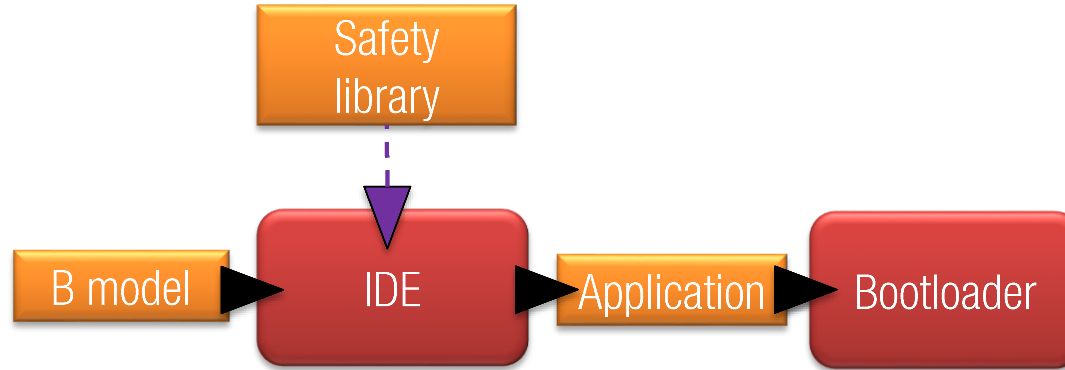


Translation from relay scheme to B

Feedback from the IDE to the initial relay scheme model
- Unsupported B construct
- Unproved proof obligation
- Any information to help the engineer to fine tune the DSL model



Roadmap



Improved, more resilient IDE

Better integration with the application generation toolchain

Improved proof performances

CLEARSY
SYSTEMS ENGINEERING

Roadmap

Application → Bootloader

LCHIP board
µC1
µC2

Inputs (10x)
Outputs (6x)
Digital I/O

LCHIP board
µC1
µC2

Inputs (10x)
Outputs (6x)
Digital I/O

LCHIP board
µC1
µC2

Inputs (10x)
Outputs (6x)
Digital I/O

Local network

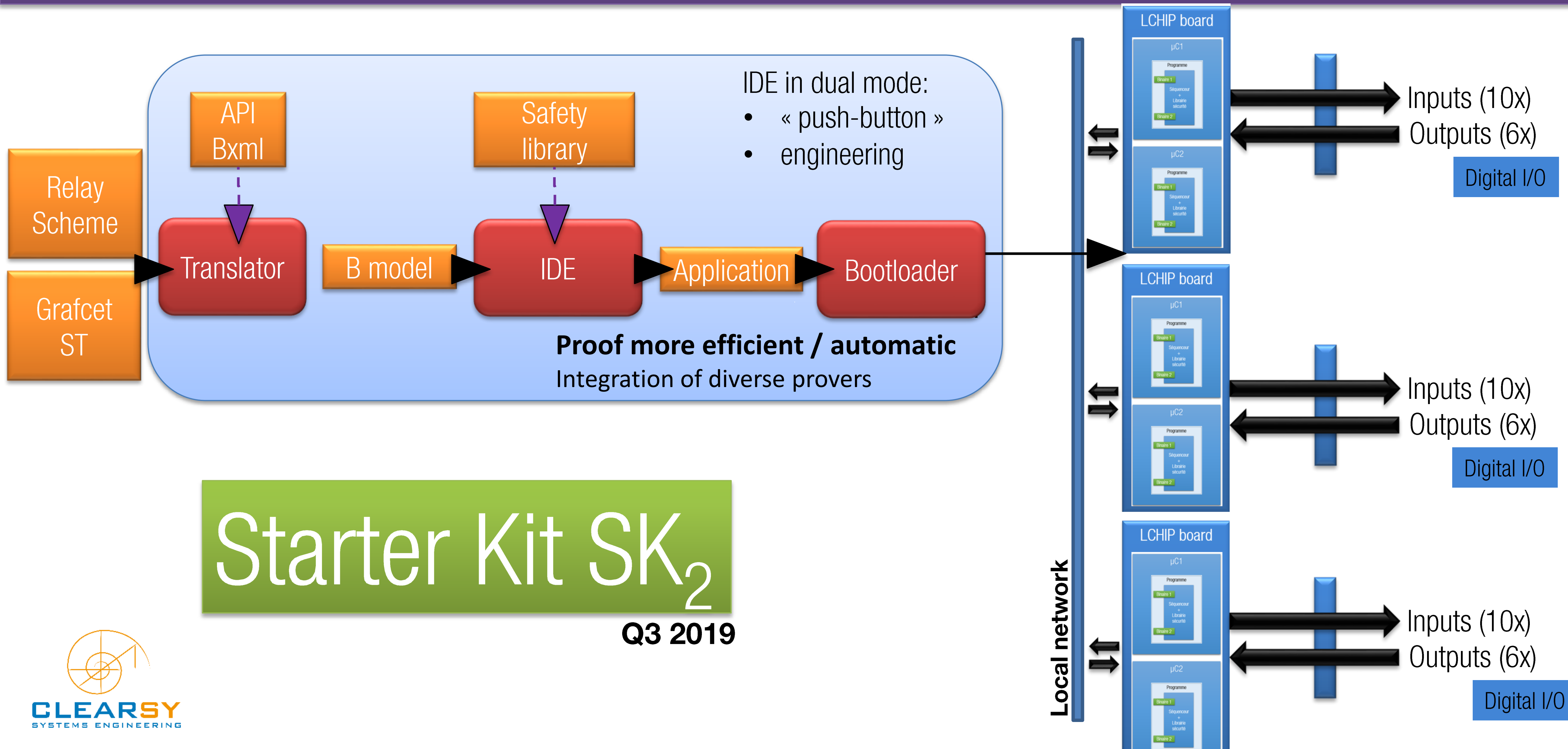
More I/O per board

Boards to connect on a local network to exchange data

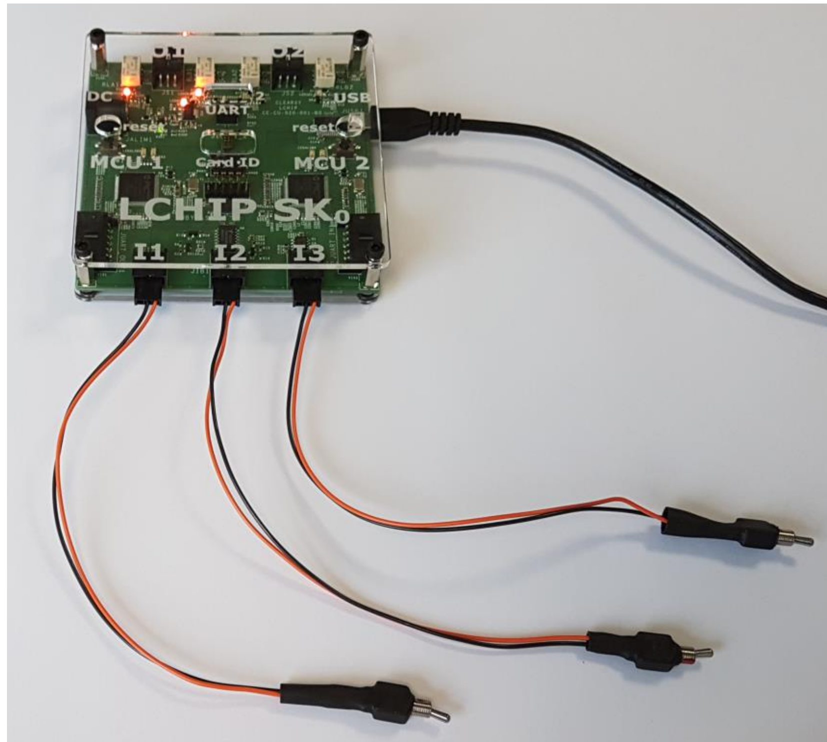
Every LCHIP board compute the overall user_logic related to all IO

B model generated from configuration data

Roadmap

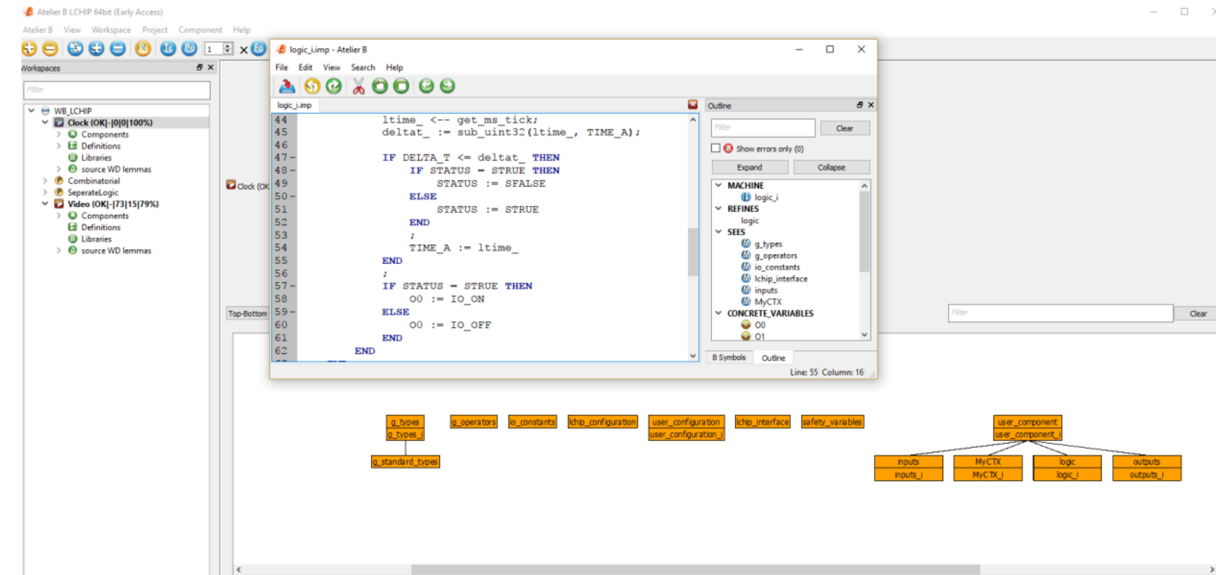


Starter Kit SK_0

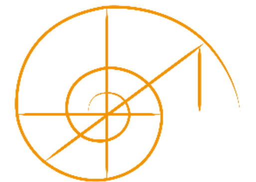


USB-C connector
Power, upload, monitor

Switches
Simulate digital inputs



Atelier B 4.4 LCHIP Early access
Model, prove, compile, upload



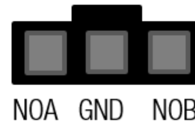
CLEARSY
SYSTEMS ENGINEERING

Starter Kit SK_0

Typical usage

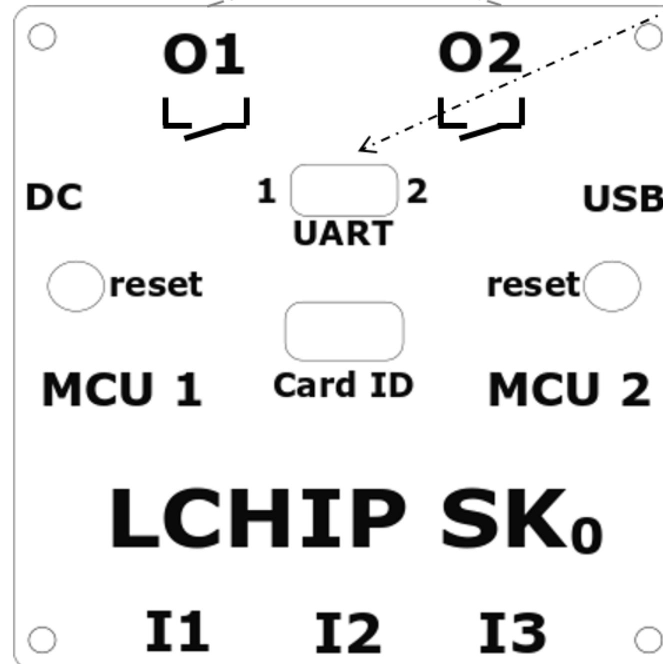
State contact read

Output connector



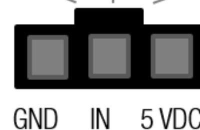
UART selection for monitoring via USB

5 VDC 500 mA for external supply



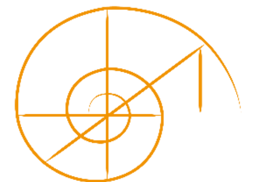
USB connector for upload and monitor

Input connector



Typical usage

Input 5 VDC come from LCHIP card supply
Maximum cutting power of output is 0.3 A at 125 VAC, 1 A at 30 VDC



CLEARSY
SYSTEMS ENGINEERING

A complete transaction

- Installing the IDE
- Starting the IDE
- Creating a project
- Importing the LCHIP infrastructure
- Contributing
- Proving
- Compiling
- Uploading
- Monitoring

Copy the the content of the USB in a directory that contains no space nor no special character. Execute Register LCHIP.cmd

Execute startAB.cmd. For the first execution, change Atelier B/preferences/langage to english/us. Quit then restart to get the english GUI

Click the yellow + button, select software development, give a name, press next then finish. Double-click on the project name to open it.

Select an open project. Select Import LCHIP API. Press Fermer. Press F5 to update the project status to see added components.

Double-click on a component. Modify it. Ctrl+S to save. Select the component then press the blue TC button to operate the typecheck.

Select the component then press the blue F0 button to operate the proof.

Select the LCHIP project then select compile LCHIP.

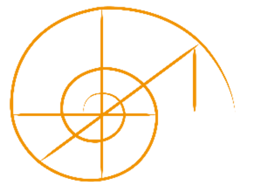
Select upload. Select connect on the uploader window. Select Erase-Program-Verify. Trigger one reset button on the board. When the device is ready, trigger one reset button to leave bootload mode.

Select monitor on the uploader window. Disconnect to stop monitoring.

CLEARSY
SYSTEMS ENGINEERING

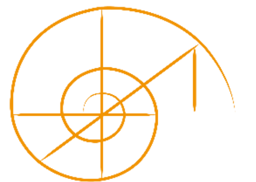
A complete transaction

VIDEO



CLEARSY
SYSTEMS ENGINEERING

Programming

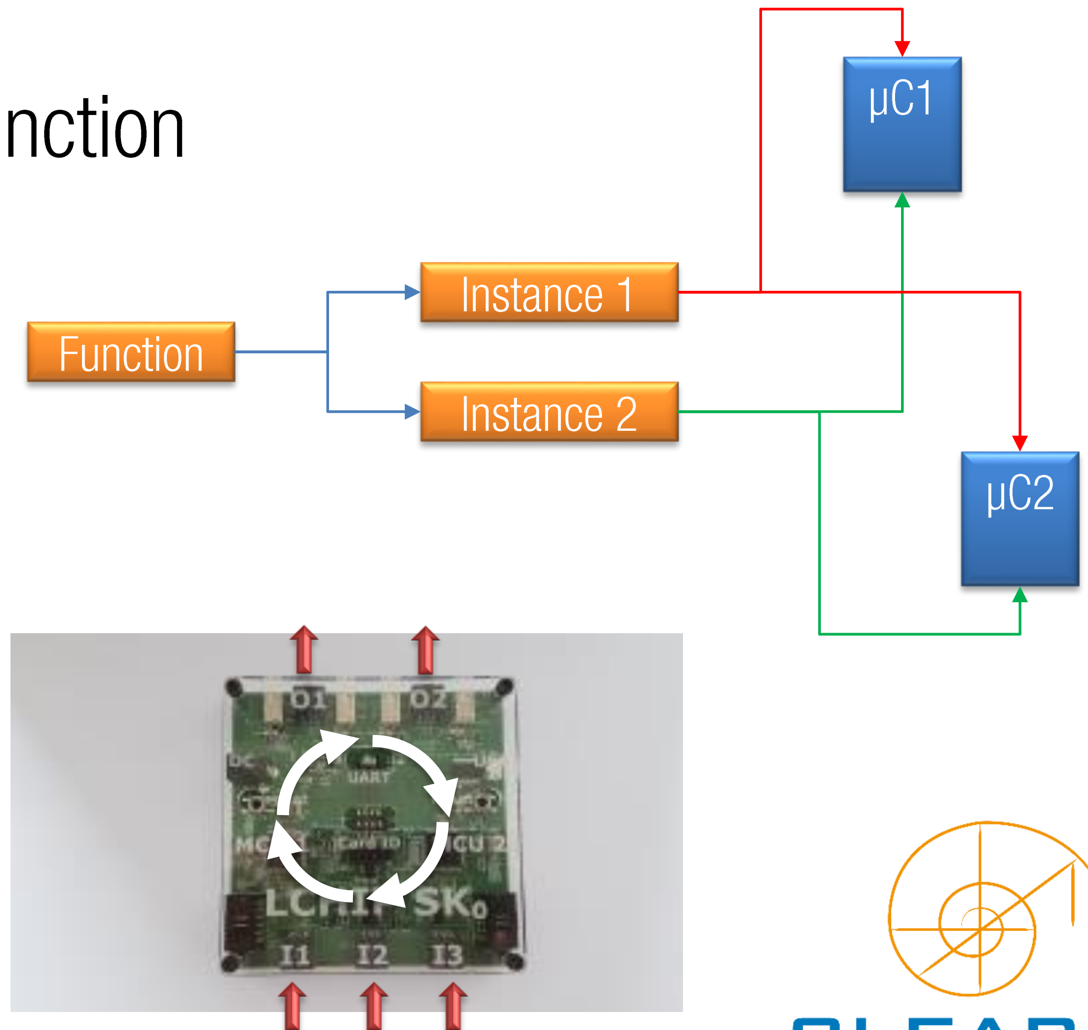


The programming model

The developer is focused on developing a function

- Independently of its transformation and distributed execution

- Can read inputs
- Can perform computations using a subset of the B language
- Can modify the outputs
- Can read the current time since the CU started



CLEARSY
SYSTEMS ENGINEERING

The programming model

```
init();

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

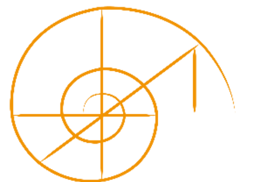
The execution is cyclic

- 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 50 ms, SK_0 enters panic mode)
- No delay()

- Inputs are values captured at the beginning of a cycle (instantaneous values)
- Outputs are maintained from one cycle to another



CLEARSY
SYSTEMS ENGINEERING

The programming model

LCHIP takes care of all safety verifications

If a divergent behaviour is detected

- one of the 4 software instances behaves differently
- one UC behaves differently

or if a structural error is detected

- bad CRC on memory
- UC unable to execute an operation properly
- etc.

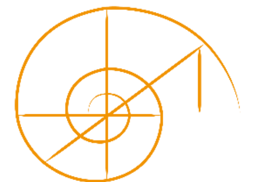
then the detecting UC reboots

Verification intra-MCU
13 000 per second

Verification inter-MCU
48 per second

measured on Clock example

Verification deferred
over several cycles



A LCHIP Project

A LCHIP project is a B project

- is generated automatically from board configuration (# IOs, naming)
- in SK_0, more information than required is exposed

It contains

g_types / g_types_i

g_standard_types

g_operators

io_constants

lchip_configuration

user_configuration / user_configuration_i

lchip_interface

safety_variables

The interface with the safety library

user_component / user_component_i

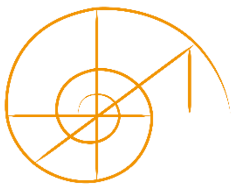
inputs / inputs_i

logic / logic_i

outputs / outputs_i

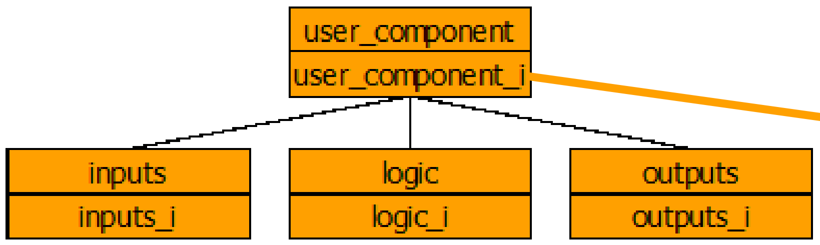
Components to modify

The model of the function



CLEARSY
SYSTEMS ENGINEERING

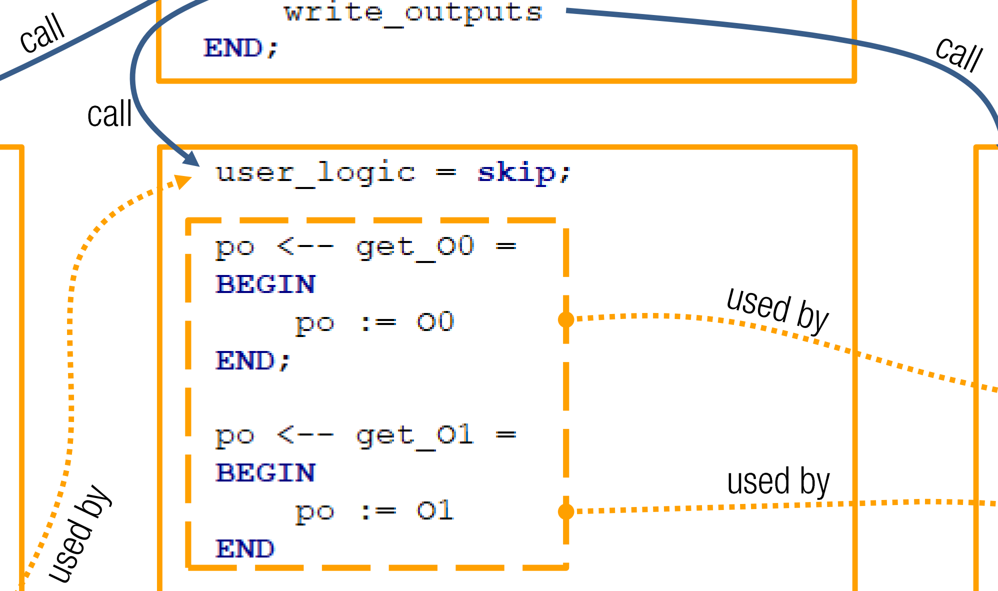
Function model




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

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

```
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
```




```
user_logic = skip;

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

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

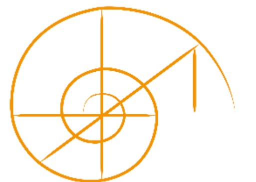
```
write_outputs =
VAR
    lsb
IN
    lsb:(lsb : uint8_t);

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

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

Programming LCHIP is mainly implementing *user_logic* OPERATION

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

Introduction to B

- Limited to the minimum, as the modelling framework is generated
- Need to know how to fill the holes



Introduction to B

- A B project is made of models

user_component	specification
user_component_i	implementation

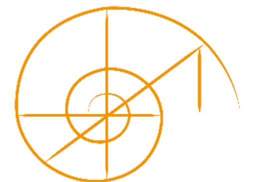
inputs	logic	outputs
inputs_i	logic_i	outputs_i

```
MACHINE
    logic

SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface

ABSTRACT_VARIABLES
    o0,
    o1

INVARIANT
    o0 : uint8_t &
    o1 : uint8_t

INITIALISATION
    o0 :: uint8_t ||
    o1 :: uint8_t

OPERATIONS
    user_logic  = BEGIN
        o0 :: uint8_t
        END;
```

Constants and read-only operations may be used

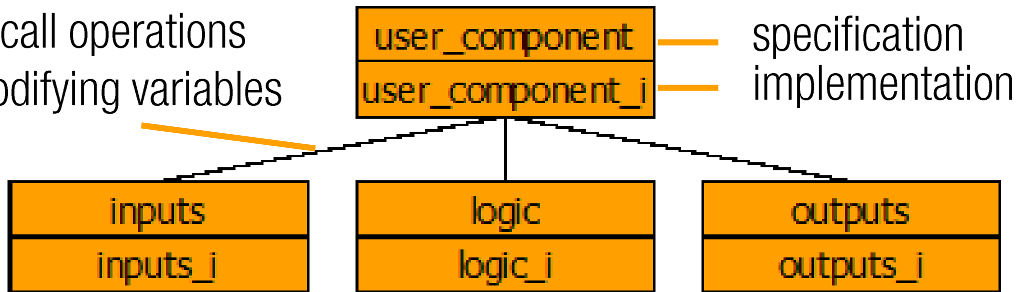
In a specification are defined:

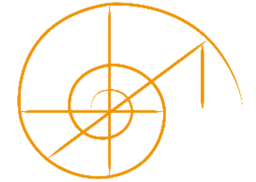
- Variables
- Properties over variables (typing + constraints)
- Initialisation (can be non-deterministic)
- Operations (can be non-deterministic)



Introduction to B

- A B project is made of models





```
IMPLEMENTATION logic_i
REFINES logic
SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface,
    inputs,
    MyCTX
    // pragma SAFETY_VARS
CONCRETE_VARIABLES
    O0,
    O1,
    TIME_A, // Time of the latest update
    STATUS // Clock up or down
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
OPERATIONS
    user_logic =
    BEGIN
        O0 := IO_OFF;
        O1 := IO_OFF
    END;
```

Variables are valued

Algorithm using substitutions

In an implementation are defined:

- Variables
- Properties over variables (typing + constraints)
- Initialisation (is deterministic)
- Operations (are deterministic)

Introduction to 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

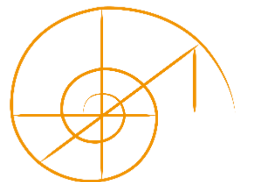
```
// pragma SAFETY_VARS
```

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
```



CLEARSY
SYSTEMS ENGINEERING

Introduction to B: constants declaration

specification

```
CONCRETE_CONSTANTS
    DELTA_T

PROPERTIES
    DELTA_T : uint32_t
```

implementation

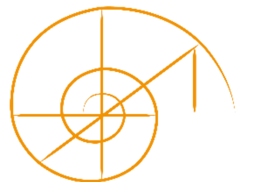
```
// pragma CONSTANTS
```

Mandatory
Contains constants that will be verified

Important
A model cannot contain both variables and constants

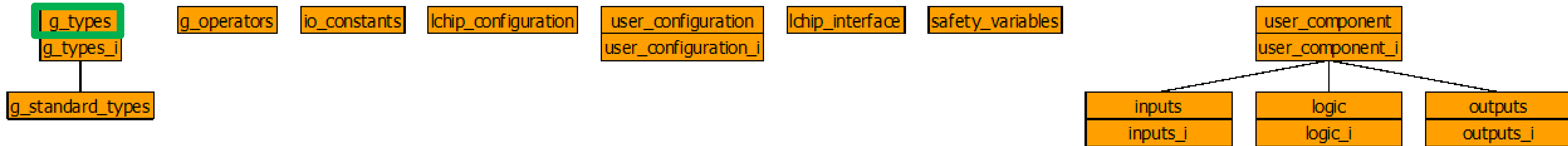
```
VALUES
    DELTA_T = 1000 // 1000 ms == 1s
```

Value that should enforce the properties



CLEARSY
SYSTEMS ENGINEERING

Introduction to B: 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

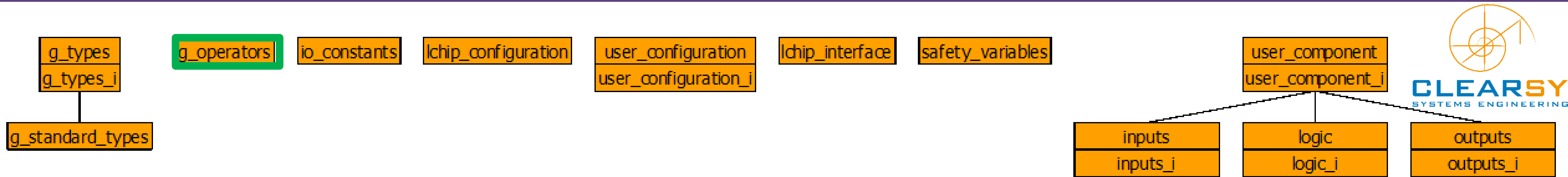
Boolean values TRUE and FALSE coded on 8 bits

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 &
```

CLEARSY
SYSTEMS ENGINEERING

Introduction to B: 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
```

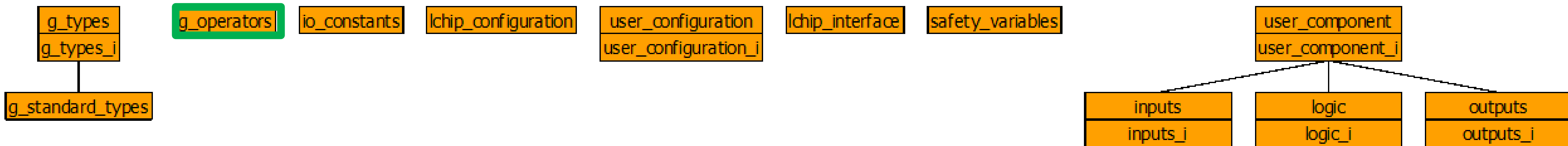
Builtin operators

```
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

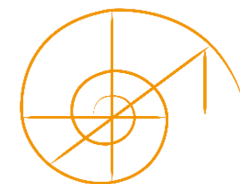
Introduction to B: unsigned int operators




```
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
```

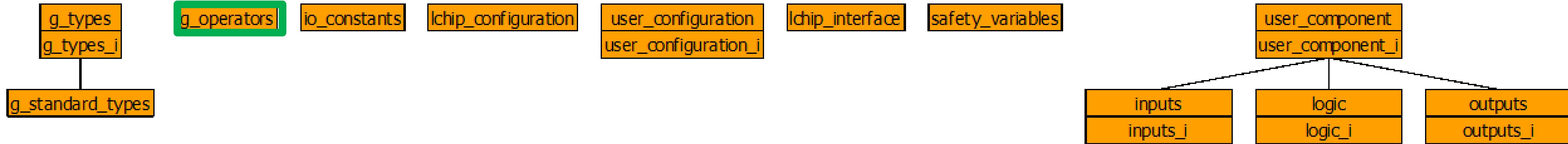
Builtin operators

```
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 &
```



CLEARSY
SYSTEMS ENGINEERING

Introduction to B: unsigned int operators



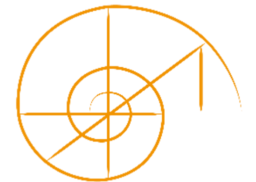

```
add_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 + x2) mod (MAX_UINT32 + 1)) &
sub_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 - x2 + MAX_UINT32 + 1) mod (MAX_UINT32 + 1)) &
mul_uint32 = % (x1, x2) . (x1 : uint32_t & x2 : uint32_t | (x1 * x2) mod (MAX_UINT32 + 1)) &
add_uint16 = % (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

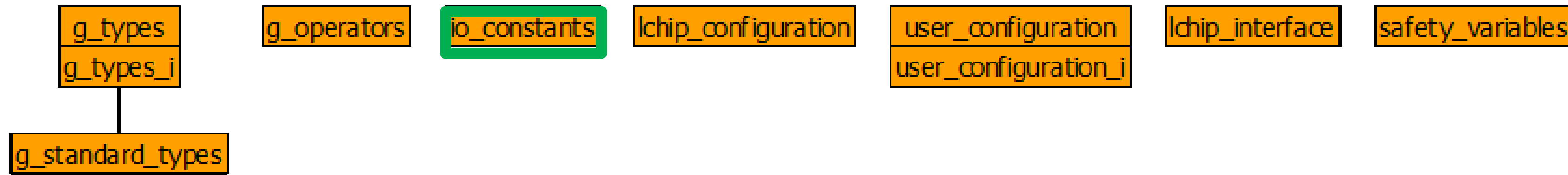
that takes two 32-bit unsigned integer parameters

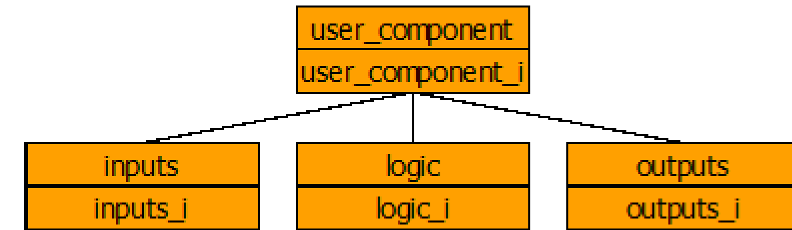
and returns the sum of the values modulo MAX_UNIT32 +1



CLEARSY
SYSTEMS ENGINEERING

Introduction to B: 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
```

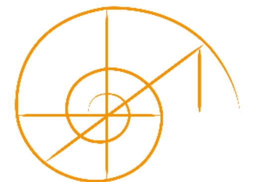
inputs and outputs state

values used by digital inputs and outputs

coded on 8 bits

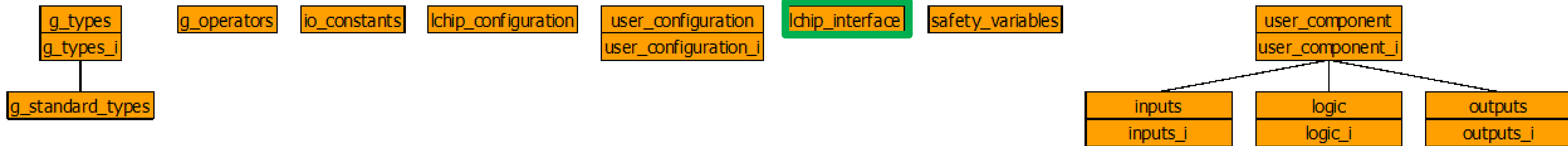
Verification

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



CLEARSY
SYSTEMS ENGINEERING

Introduction to B: inputs/outputs

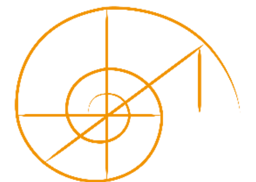



```
out <-- get_ms_tick =
PRE
    out : uint32_t
THEN
    out := ms_tick
END
```

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



Introduction to 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)

CLEARSY
SYSTEMS ENGINEERING

Introduction to 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

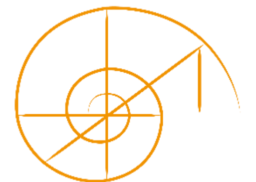
```
user_logic =
BEGIN
    VAR time_ IN
        time_ : (time_ : uint32_t);
        time_ <-- get_ms_tick;
        IF 2000 <= time_ THEN
            O1 := IO_ON
        END
    END
END;
```

Local variables declaration

Operation call

Important

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



CLEARSY

SYSTEMS ENGINEERING

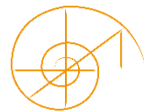
Introduction to B: user_logic

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
```

Introduction to B: 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

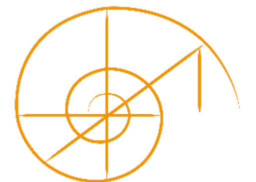
« then » is related to the valuation of O0 regarding O1
O0 and O1 will be positioned at the same time
at the end of the cycle

CLEARSY
SYSTEMS ENGINEERING

Troubleshooting

Verification process

- When a model is modified, Ctrl+S to save.
  - Check if the outline is OK or contains error messages
  - Check if some parts of the model are not underlined in red
- When the model is saved OK, select the model and hit TC blue button (typecheck)
  - Check if errors are displayed in the Atelier B main window
- When the model is typechecked OK, select the model and the F0 blue button (proof inforce 0)
  - Check if the column « unproved » is 0 (if not, nothing else to do at the moment - wait for SK_1)
- Compile the project. In case of failure
  - Check logs for more info. Ask for help.
- Upload the project. In case of failure
  - Check the common sources of error. Ask for help.
- Execute the function. In case of failure
  - Check the common sources of error. Ask for help.



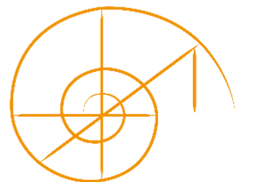
CLEARSY
SYSTEMS ENGINEERING

Troubleshooting

Common sources of error

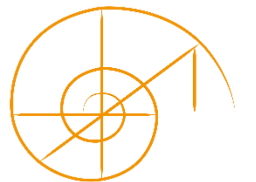
- Using non-supported operators: +, -, *, /
- Using non-supported types: INT, NAT, BOOL, STRING
- Writing a 32-bit unsigned int into a 16-bit or 8-bit
- Writing a 16-bit unsigned int into a 8-bit
- Allocating too much memory: table 49k of uint8_t == 100% memory
- Too many computations preventing inter-MCU verifications (browsing 29k cells of a table)
- Hint: Have a look at the logs on dcc_build/log

- Bxml error: unsupported language
- Panic mode:
  - memory problem,
  - transfer interrupted (CRC),
  - unable to perform MCU verification on time



CLEARSY
SYSTEMS ENGINEERING

Use case #1: combinatorial hello world



UC1: combinatorial

$O_0 = I_0$ and I_1 and I_2
$O_1 = not(O_0)$

```
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

Local variables are valued with LOCAL_OPERATIONS

UC1: combinatorial

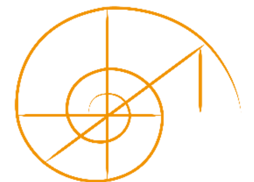
```
LOCAL_OPERATIONS
    res <-- triAND(v1, v2, v3) =
    PRE
        v1: uint8_t &  v2: uint8_t & v3: uint8_t
    THEN
        res :: uint8_t
    END
    ;
    res <-- negIO(val) =
    PRE
        val : uint8_t
    THEN
        res :: uint8_t
    END
```

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 type first



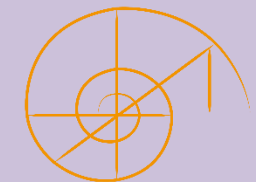
UC1: combinatorial

To do:

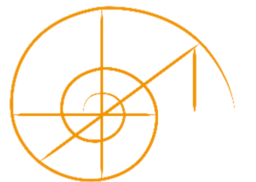
- Open the Combinatorial project
- Prove it (select all the components then F0)
- Check the numbers (the table shown in the « classical view »)
- Compile it
- Upload it
- Execute it
- Monitor it
- Play with the switches to verify the behavior

To do next:

- Improve the specification of user_logic in the logic component
- Improve the specification of the local operations triAnd and negIO
- $O_0 = I_0$ or I_1 or I_2



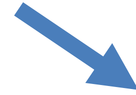
Use case #2: clock



CLEARSY
SYSTEMS ENGINEERING

UC2: clock

$O_0 = not(O_0)$ every 1 second

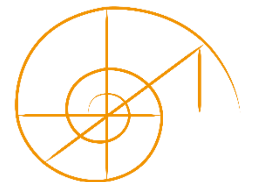


Current time

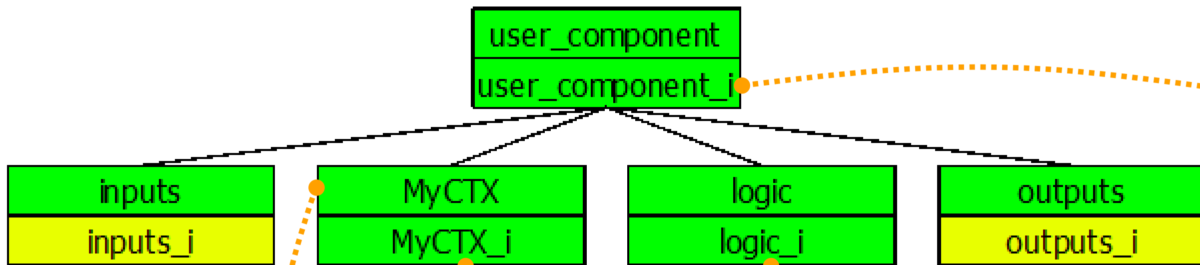
User defined constant

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

        ltime_ <-- get_ms_tick;
        deltat_ := sub_uint32(ltime_, TIME_A);

        IF DELTA_T <= deltat_ THEN
            IF STATUS = STRUE THEN
                STATUS := SFALSE
            ELSE
                STATUS := STRUE
            END
            ;
            TIME_A := ltime_
        END
        ;
        IF STATUS = STRUE THEN
            O0 := IO_ON
        ELSE
            O0 := IO_OFF
        END
    END
END;
```

STRUE and SFALSE are logical values coded on 8 bits uint



UC2: clock




```
IMPLEMENTATION user_component_i

REFINES user_component

SEES
    g_types

IMPORTS
    inputs,
    logic,
    outputs,
    MyCTX
```

```
MACHINE
    MyCTX
CONCRETE_CONSTANTS
    DELTA_T
SEES
    g_types
PROPERTIES
    DELTA_T : uint32_t

END
```

Read access to g_types for uint32_t declaration

```
IMPLEMENTATION MyCTX_i
REFINES MyCTX

// pragma CONSTANTS

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

Contains constants

```
IMPLEMENTATION logic_i

REFINES logic

SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface,
    inputs,
    MyCTX
```

CLEARSY
SYSTEMS ENGINEERING

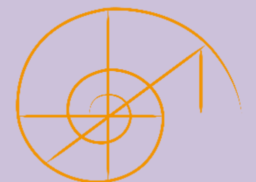
UC2: clock

To do:

- Open the Clock project
- Prove it (select all the components then F0)
- Check the numbers (the table shown in the « classical view »)
- Compile it
- Upload it
- Execute it
- Monitor it

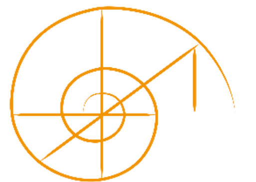
To do next:

- $O_1 = not(O_1)$ every 2 seconds
- Imagine a more precise version of a clock (O_0 original clock, O_1 improved clock)
(hint: the current version calculates a delay since last tick and not since the beginning)



CLEARSY
SYSTEMS ENGINEERING

Use case #3: tables & loops

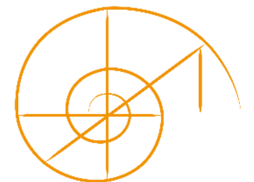


UC3: tables & loops

```
CONCRETE_VARIABLES
    O0,
    O1,
    Table8,
    CurrentIndex

INVARIANT
    O0 : uint8_t &
    O1 : uint8_t &
    Table8 : 0..10 --> uint8_t &
    CurrentIndex : uint8_t

INITIALISATION
    O0 := IO_OFF;
    O1 := IO_OFF;
    Table8 := (0..10) * {0};
    CurrentIndex := 0
```

} A table is a total function. Table indexes always start at position 0
Here a table of 11 elements of 8 bits uint

} Initialisation of the complete table with the value 0



CLEARSY
SYSTEMS ENGINEERING

UC3: tables & loops

```
user_logic =
BEGIN
// Table8 := (0..10) * {0}
    VAR index_ IN
        index_ : (index_ : uint8_t);

        index_ := 0;

        WHILE index_ <= 10 DO
            Table8(index_) := 0;
            index_ := add_uint8(index_, 1)
        INVARIANT
            index_ : 0..11
        VARIANT
            11-index_
        END
    END

END;
```

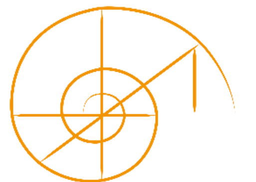
Loop index

Body of the loop

Values taken by the index during the iterations

Integer value that is decreased at each iteration
When 0 is reached, the loop terminates

In B, loops are finite and always terminate



CLEARSY
SYSTEMS ENGINEERING

UC3: tables and loops

To do:

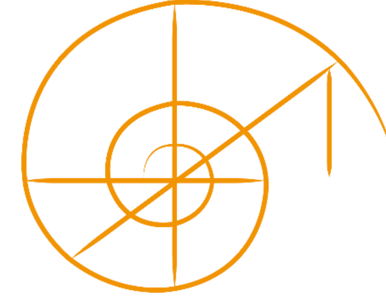
- Create a project
- Create a table where you store successive values of the inputs.
  Regularly check if this history contains 3 inputs up at the same time more than once.
  If yes, activate one output.
  Clear the history. Do it again.
- Prove it (select all the components then F0)
- Check the numbers (the table shown in the « classical view »)
- Compile it, Upload it, Execute it, Monitor it

To do next:

- IO_ON & IO_OFF are two uint8_t constant values that encode safely the two states. Imagine a solution to output these two values

LCHIP

LOW-COST
HIGH-INTEGRITY
PLATFORM



CLEARSY
SYSTEMS ENGINEERING

Thank you for your attention

Pistoia, November 14, 2017
Thierry Lecomte & Patrick péronne, ClearSy
{thierry.lecomte, patrick.peronne}@clearsy.com

LCHIP is the name of a French R&D project (FUI21) funded by Bpifrance, the public bank for innovation

Funded by

Région
Provence
Alpes
Côte d'Azur

MÉTROPOLE
AIX-MARSEILLE
PROVENCE