Programming the CLEARSY Safety Platform with B

Thierry Lecomte(✉)

ClearSy, 320 Avenue Archimède, Aix en Provence, France
thierry.lecomte@clearsy.com

Abstract. The CLEARSY Safety Platform (CSSP) is aimed at easing the development and the deployment of safety critical applications, up to the safety integrity level 4 (SIL4). It relies on the smart integration of the B formal method, redundant code generation and compilation, and a hardware platform that ensures a safe execution of the software. This paper exposes the programming model of the CSSP used to develop control & command applications based on digital I/Os.

Keywords: B method · Safety critical · Programming model

1 Introduction

In many industrial standards, formal methods are highly recommended when developing safety critical software for the highest safety levels. However formal methods are highly recommended just like many other non-formal (combination of) techniques, as these recommendations are setup collectively and represent the industrial best practices. Convinced that formal methods could help to obtain better products [4,5,7,8], more easily certifiable, a generic, safe execution platform has been researched for years, combining safety electronics and defect-free proven software. The software model is proved to be defect-free - complying with its formal specification and without programming errors. The code generators and the compilers are not defect-free. They are not required to be defect-free as the defects are detected with divergent behaviour during execution. The CLEARSY Safety Platform was initially an in-house development project before being funded by the R&D collaborative project *LCHIP* (Low Cost High Integrity Platform) to obtain a generic version of the platform (i.e. not only aimed at railway systems). *LCHIP* [6] is aimed at allowing any engineer to develop a function by using its usual Domain Specific Language (DSL) and to obtain this function running safely on a hardware platform. With an automatic development process, the B formal method will remain "behind the curtain" in order to avoid expert transactions over several languages (domain specific language, B language, interactive proof). Indeed the programs developed with the CLEARSY Safety Platform are considerably simpler than metro automatic pilot, with few properties, simpler algorithms and hence with an expected excellent automatic proof ratio. The integration of third party provers/solvers is also

A. Raschke et al. (Eds.): ABZ 2020, LNCS 12071, pp. 124–138, 2020.
https://doi.org/10.1007/978-3-030-48077-6_9

expected to improve automatic proof. Based on our previous certification experience, the safety demonstration of a safety case does not require any specific feature for the input B model; it could be handwritten or the by-product of a translation process. Several DSLs are being connected (or planned to be) based on an Open API (Bxml).

This paper introduces the CLEARSY Safety Platform, presents and explains the evolution of the supported B0 modelling language. The shape of the programs developed for this platform are tightly linked with the specific mission of the platform: ensuring a safety (see Sect. 3.3) out of reach of the developer who cannot alter it.

This paper is structured in five parts. The Terminology is first introduced as some terms and concepts are quite specific. Then a description of the CLEARSY Safety Platform is provided with a focus on its safety features. Third the programming model is introduced; the simplification of the proof is also discussed. Exploitation and dissemination are then exposed. Finally conclusion and perspectives are discussed.

2 Terminology

This chapter clarifies a number of unusual terms and concepts used in this paper.

Atelier CSSP is Atelier B extended with diverse code generator toolchain, bootloader, and a new project type (CSSP project).

B0 is a subset of the B language [1] that must be used at implementation level. It contains deterministic substitutions and concrete types. B0 definition depends on the target hardware associated to a code generator [2]. Most railways product lines use their own own specific code generator.

Bxml is an XML interface to B models, supported by Atelier B.

CRC stands for cyclic redundancy check, is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

CSSP abbreviation of CLEARSY Safety Platform. The CLEARSY Safety Platform is made up of a hardware execution platform, an IDE enabling the generation of diverse binaries from a single B model, and a certification kit describing its safety features as well as the safety constraints exported to the hosting system.

Diversity intentional differences between redundant components, to reduce the likelihood of common failures due to systematic causes that would reduce the benefit of redundancy [3].

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its components. In our case, any electronic part including the processors.

HEX is a file format that conveys binary information in ASCII text form. It is commonly used for programming microcontrollers, EPROMs, and other types of programmable logic devices.

PLC stands for programmable logic controller, is an industrial digital computer which has been ruggedized and adapted for the control of any activity that requires high reliability control and ease of programming and process fault diagnosis.

Safety refers to the control of recognized hazards in order to achieve an acceptable level of risk.

SIL put for Safety Integrity Level, is a relative level of risk-reduction provided by a safety function. Its range is usually between 0 and 4, SIL4 being the most dependable and used for situations where people could die.

Reliability is the ability of a system to perform its required functions under stated conditions for a specified time.

3 The CLEARSY Safety Platform

3.1 Rationale

Developing a safety computer from scratch is not something you easily decide because of the effort required to obtain such a device. Two kinds of device are currently available on the market for safety critical applications: PLCs and SIL3/SIL4-ready boards. Large companies building trains have their own in-house devices but they are not publicly available. PLCs provide a strict, certified environment from which it is impossible to escape, requiring systems to be designed and programmed in specific ways. On the contrary, SIL3/SIL4-ready boards offer more freedom, come with hardware features not incompatible with the standards but where the safety principles have to be fully programmed by the developer in C or similar language.

To overcome this inconvenience, CLEARSY decided to develop its own solution based on the combination of redundant hardware and proven software developed with B. Producing its own hardware would reduce by an order of magnitude its cost compared to PLCs and SILx-ready boards while using Atelier B would allow more freedom and more control on the software development. The decision to go for B was easily taken as it is highly recommended by the industry standard for SIL4 software development. B is also the central formal technology we have been using during more than 20 years for most of safety critical software development. Finally the CLEARSY Safety Platform is aimed at easing the certification process, as the safety principles, embedded in the electronics design and the B software, are out of reach of the developer who cannot alter them.

3.2 Description

The CLEARSY Safety Platform (abbreviated as CSSP in the rest of the document) is a new technology, both hardware and software, combining a software development environment based on the B language and a secured execution hardware platform, to ease the development of safety critical applications.

It relies on a software factory that automatically transforms function into binary code that runs on redundant hardware. The starting point is a text-based, B formal model that specifies the function to implement. This model may contain static and dynamic properties that define the functional boundaries of the target software. The B project is automatically generated (Fig. 5), based on the inputs/outputs configuration (numbers, names). The project contains all the machines and implementation components required to program the CLEARSY Safety Platform. From the developer's point of view, only one function (name *user_logic*) has to be specified (machine *logic*) and implemented properly (implementation *logic*_i).

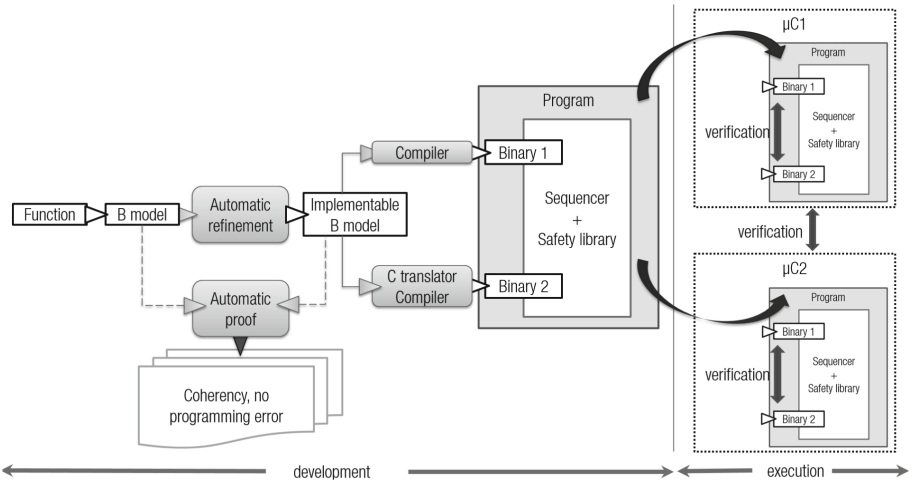

Fig. 1. The safe generation and execution of a function on the double processor.

The implementable model is then translated using two different chains:

- Translation into C ANSI code, with the C4B Atelier B code generator (instance I_1). This C code is then compiled into HEX[1] binary code with an off-the-shelf compiler (gcc).
- Translation into MIPS Assembly then to HEX binary code, with a specific compiler developed for this purpose (instance I_2). The translation in two steps

[1] A file format that conveys binary information in ASCII text form. It is commonly used for programming micro-controllers.

allows to better debug the translation process as a MIPS assembly instruction corresponds to a HEX line.

The software obtained is the uploaded on the execution platform to be executed by two micro-controllers (Fig. 2).

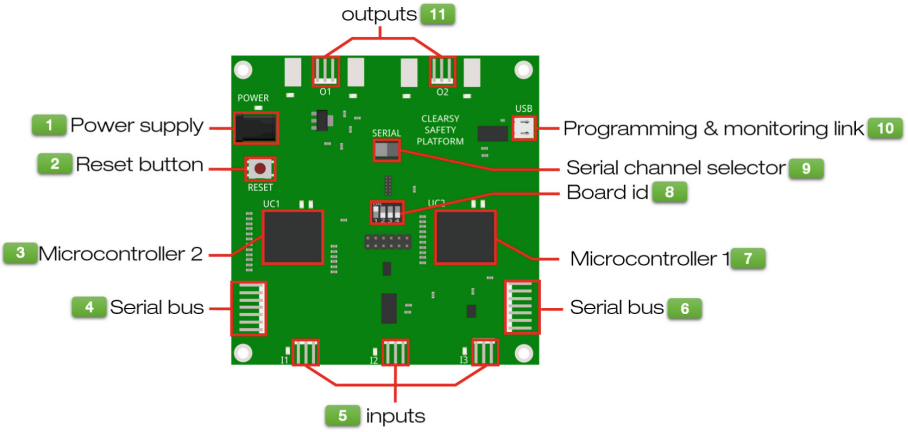

Fig. 2. The CLEARSY Safety Platform Starter Kit 0 (SK_0) – documentation available at https://github.com/CLEARSY/CSSP-Programming-Handbook

3.3 Safety

These two different instances I_1 and I_2 of the same function are then executed in sequence, one after the other, on two PIC32 micro-controllers. Each micro-controller hosts both I_1 and I_2, so at any time 4 instances of the function are being executed on the micro-controllers. The results obtained by I_1 and I_2 are first compared locally on each micro-controller then they are compared between micro-controllers by using messages. In case of a divergent behaviour (at least one of the four instances exhibits a different behaviour), the faulty micro-controller reboots. The sequencer and the safety functions are developed once for all in

```
initialisation

while(true) {
        execute I1
        execute I2
        perform safety verifications
}
```

Fig. 3. The pseudo-code of the sequencer.

B by the IDE design team and come along as a library. This way, the safety functions are out of reach of the developers and cannot be altered. The safety is based on several features such as:

- the detection of a divergent behaviour,
- micro-controller liveness regularly checked by messages,
- the detection of the inability for a processor to execute an instruction properly[2],
- the ability to command outputs[3],
- memory areas (code, data for the two instances) are also checked (no overlap, no address outside memory range),
- each output needs the two micro-controllers to be alive and providing respectively power and command, to be active (permissive mode). In case of misbehaviour, the detecting micro-controller deactivate its outputs and enter an infinite loop doing nothing.

The code generators are different (code generation paths, specification, programming languages, development teams) and as such common failure modes are neglected. Some of the tools part of the tool-chain have been "certified by usage" since 1998 (B parser, B compiler, C code generator), but the newest tools of this tool-chain have no history to rely on for certification. It is not a problem for railway standards as the whole product is certified (with its environment, its development and verification processes, etc.), hence it is not required to have every tool certified. Instead the main feature used for the safety demonstration is the detection of a misbehaviour among the 4 instances of the function and the 2 microcontrollers. This way, similar bugs that could affect at the same time and with the same effects two independent tools are simply neglected. In its current shape, the CLEARSY Safety Platform provides an automatic way of transforming a proven B model into a program that safely executes on a redundant platform while the developer does not have to worry about the safety aspects.

3.4 Target Applications

The execution platform is based on two PIC32 micro-controllers[4]. The processing power available is sufficient to update 50k interlocking Boolean equations per second, compatible with light-rail signalling requirements. The execution platform can be redesigned seamlessly for any kind of mono-core processor if a higher level of performance is required.

[2] All instructions are tested regularly against an oracle.
[3] Outputs are read to check if commands are effective, a system not able to change the state of its outputs has to shutdown.
[4] PIC32MX795F512L providing 105 DMIPS at 80 MHz.

The IDE provides a restricted modelling framework for software where:

- No operating system is used.
- Software behaviour is cyclic (no parallelism).
- No interruption modifies the software state variables.
- Supported types are Boolean and integer types (and arrays of).
- Only bounded-complexity algorithms are supported (the price to pay to keep the proof process automatic).

4 Programming Model

Target CSSP applications are controllers. They execute the following infinite loop: read inputs, perform computation, then set outputs. If a failure happens, the board deactivates the outputs (they are all OFF – not powered) and enters an infinite loop doing nothing (Fig. 4). The only way to exit this loop is to reset the board. The program in Flash memory is copied into RAM and then its execution starts. If the failure is permanent, the board keeps restarting with the outputs deactivated – the board remains in a safe, restrictive state.

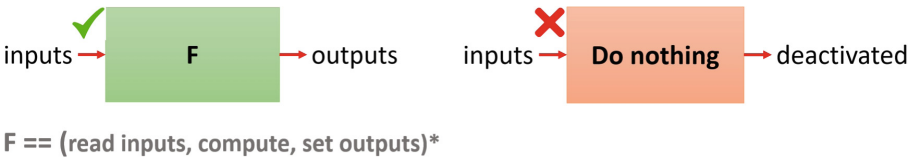

Fig. 4. A CSSP is either able to execute its software properly (transfer function F) (left) or is not able (right) and hence does nothing while its outputs are deactivated.

4.1 Development Process

A CSSP project (Fig. 5) is a B project generated from a CSSP board configuration where I/O are selected (some inputs/outputs pins may not be used) and named. This generated B project is made of:

- the interface with the safety library, containing the definition of all the types (and related constants) that may be used in a CSSP project, as well as specific operators (arithmetic, logic) and operations (access to current time, message to print on serial channel),
- the model of the function to program, that has:
  - a read-only access to the safety library, the digital inputs status (OFF, ON), the current time since the last rest/power-on, and
  - the ability to modify the digital outputs (OFF, ON).

Programming the CSSP consists in modifying the components *user_ctx* and *logic*, and to possibly add other components to be imported by *logic_i*.

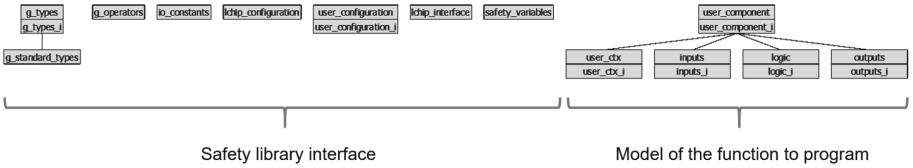

Fig. 5. A CSSP project.

4.2 Pragmas

A component cannot contain both constants (SETS, CONSTANTS) and variables. Constants are hosted by context machines (machines without variables, with possibly read-only operations). The compiler is made aware of this situation by the use of one and only one pragma in each implementation:

- CONSTANTS, to indicate a constants-only module
- SAFETY_VARS, to indicate a variables-only module

```
IMPLEMENTATION
    logic_i
REFINES
    logic
SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface,
    user_ctx,
    inputs

    // pragma SAFETY_VARS
```

```
IMPLEMENTATION
    user_ctx_i
REFINES
    user_ctx

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

Fig. 6. Two examples of pragmas.

4.3 Types and Operators

The types available in implementation are:

- *uint8_t*, *uint16_t*, *uint32_t*. These types (unsigned integers coded on 8, 16 and 32 bits) are preferred to the generic type INT, to get a better control over variable memory size and overflow. Automatic casting is performed when for example a *uint16_t* variable is combined with a *uint8_t* value. The reverse situation generates a warning from the B32 compiler.
- *BOOL*

The values of the digital inputs and outputs (*IO_OFF*, *IO_ON*) are stored as *uint8_t* and not as Boolean. It is because a memory glitch could easily transform a 0 in 1 (or a 1 in 0) without being easily detected. Having these values coded with 8 bits (with a sufficient Hamming distance) make this undetected modification unlikely to occur. Moreover setting one output with a value different from *IO_OFF* and *IO_ON* is detected during execution by the CLEARSY Safety Platform which enters panic mode.

In order to automate as much as possible the proof process, the arithmetic operators able to overflow – +, −, x – are replaced by non-overflowing operators. These operators are modelled as modulo operators (Fig. 7), preventing an overflow to happen. These operators are defined for the 3 supported arithmetic types as lambda functions and implemented with native functions in the safety library. These operators avoid to generate overflow proof obligations and enable a better automation of the proof process. However well-definedness proof obligations remain and when using the integer division/, the denominator has to be proved different from 0.

```
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)) &
```

Fig. 7. Arithmetic operators redefined.

Bitwise operators (and, or, xor, not, shift left logical, shift right logical) have been added similarly (Fig. 8). They allow programs to operate more easily at bit level. They are defined for 8, 16, and 32 bit sizes.

```
bitwise_sll_uint32 : uint32_t*uint8_t --> uint32_t &
bitwise_srl_uint32 : uint32_t*uint8_t --> uint32_t &
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_or_uint32 : uint32_t*uint32_t --> uint32_t &
```

Fig. 8. Bitwise operators added.

4.4 Time

Time is defined as a *uint32_t* and represent a number of milliseconds. The operation *get_ms_tick* returns the number of milliseconds elapsed since the last reset or power on. Storing the current time and then checking its difference with a future current time allows one to program timers.

4.5 I/O

Inputs and outputs valid values are *IO_OFF* and *IO_ON*. To get the value of an input, use the operation get_xxx where xxx is the name given to the input. The operation returns a *uint8_t*. To set the value of an output, use the operation set_xxx where xxx is the name you gave to the output.

4.6 Substitutions

The B0, implementation language, supported by the CLEARSY Safety Platform is more strict than the one supported by the C code generator C4B. The main reason for not providing as much freedom to the develop is to keep the B32 compiler simple in order to more easily convince the safety auditor during the certification process. Several substitutions are constrained as follow:

- *IF THEN ELSE* supports only single condition. If a test is a disjunction/conjunction of several expressions, the test will have to be nested into several levels. Testing operators are restricted to $<$, $\leq$ and $=$.
- assignments are restricted to two operands on the right hand term in order to avoid to manipulate the stack. The valuation with the addition of more than two operands will have to be decomposed in successive additions with two operands.
- variables declared in a VAR substitution have to be typed first with a substitution "becomes such that".

```
user_logic =
BEGIN
    VAR i1_, i2_, i3_ IN
        i1_ :( i1_ : uint8_t );
        i2_ :( i2_ : uint8_t );
        i3_ :( i3_ : uint8_t );

        i1_ <-- get_I1;
        i2_ <-- get_I2;
        i3_ <-- get_I3;

        O1 <-- triAND(i1_, i2_, i3_);
        O2 <-- negIO(O1)
    END
END;
```

```
res <-- triAND(v1, v2, v3) =
BEGIN
    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;
```

Fig. 9. Local variables in *user_logic* are types before use. Tests in *triAND* are nested because only single conditions are supported.

5 Ease to Prove Models

One of the objectives of the CLEARSY Safety Platform is to make to proof process fully automated. The use of the modulo arithmetic operators contributes

directly to this objective. The low complexity of the target lightweight applications (smaller and simpler than metro automatic pilots for example) is another reason to keep the proof effort low.

However given the modelling choices made for the arithmetic operators and the heavy use of lambda functions, we had to make sure that trivial arithmetic assignment with these operators would lead to proof obligations that are provable automatically. The analysis of the proof obligations initially that were not demonstrated automatically led to the addition of several proof elements (Fig. 10):

- two rules to handle properly any predicate containing 2**x. These rules appear in the PatchProver, a slot for mathematical rules to be applied for any project. PatchProverA means that these rules are applied after (A put for After) the one iteration of the main prover.
- several proof tactics in the User_Pass of several components.

```
PatchProverA0
  PatchProverA0.1
    x : NATURAL
    =>
    0+1<=2**x
  PatchProverA0.2
    x : NATURAL
    =>
    not(2**x = 0)
```

```
THEORY User_Pass IS
    Operation(ValuesLemmas) & ff(0) & mp & pp(rp.0 | 5) & pp(rp.0 | 5);
    Operation(ValuesLemmas) & ff(0) & dd(0) & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss &
mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp &
ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss &
mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & ss & mp & mp & ah(Mhyp(xx :
s)) & pp(rp.0 | 60) & ah(Mhyp(xx : s)) & pp(rp.0 | 60) & pp(rp.0 | 60)
END
```

Fig. 10. Mathematical rules and User_Pass proof tactic defined to automate proof.

Finally the default CSSP project generated after creation is fully proved automatically with the following scenario: select all components, prove force 0, prove user pass. It also applies for the two examples provided with the Atelier CSSP: Clock and Combinatorial (Fig. 11).

Clock (OK|OK|-|-|83|0|100%)

Classical view

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved
g_operators	OK	OK	37	37	0
g_standard_types	OK	OK	0	0	0
g_types	OK	OK	3	3	0
g_types_i	OK	OK	10	10	0
inputs	OK	OK	0	0	0
inputs_i	OK	OK	6	6	0
io_constants	OK	OK	0	0	0
lchip_configuration	OK	OK	0	0	0
lchip_interface	OK	OK	1	1	0
logic	OK	OK	0	0	0
logic_i	OK	OK	6	6	0
outputs	OK	OK	0	0	0
outputs_i	OK	OK	4	4	0
safety_variables	OK	OK	0	0	0
user_component	OK	OK	0	0	0
user_component_i	OK	OK	2	2	0
user_configuration	OK	OK	0	0	0
user_configuration_i	OK	OK	13	13	0
user_ctx	OK	OK	0	0	0
user_ctx_i	OK	OK	1	1	0

Combinatorial (OK|OK|-|-|76|0|100%)

Classical view

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved
g_operators	OK	OK	37	37	0
g_standard_types	OK	OK	0	0	0
g_types	OK	OK	3	3	0
g_types_i	OK	OK	10	10	0
inputs	OK	OK	0	0	0
inputs_i	OK	OK	6	6	0
io_constants	OK	OK	0	0	0
lchip_configuration	OK	OK	0	0	0
lchip_interface	OK	OK	1	1	0
logic	OK	OK	0	0	0
logic_i	OK	OK	0	0	0
outputs	OK	OK	0	0	0
outputs_i	OK	OK	4	4	0
safety_variables	OK	OK	0	0	0
user_component	OK	OK	0	0	0
user_component_i	OK	OK	2	2	0
user_configuration	OK	OK	0	0	0
user_configuration_i	OK	OK	13	13	0
user_ctx	OK	OK	0	0	0
user_ctx_i	OK	OK	0	0	0

Fig. 11. Both project Clock and Combinatorial, provided with Atelier CSSP are fully proved automatically with added rules and predefined tactics.

Of course, the added rules and tactics are not sufficient to automatically prove all the proof obligations generated for the CSSP but provide a basis for reuse and extension, together with existing mathematical rules (including packages s1 and b1, added after the end of the development of the automatic metro line 14 in Paris, and able to simplify arithmetic predicates and expressions).

6 Reaching the Limits

The CSSP is intrinsically different from an Arduino as its offers safety features. However if these safety features cannot be demonstrated, a CSSP is not distinguishable from an Arduino. The following situations allow to demonstrate some of the safety features:

- **2oo2 principle:** corrupting the memory is not easily performed as it requires generating perturbing electromagnetic field and some luck to indeed modify the memory. Instead the CSSP software interface provides two functions, *get_instance_id()* and *get_processor_id()*, which allow to program a behaviour dependent on the software instance and on the processor executing the software. In this case, a divergent behaviour could be obtained leading to the panic mode.
- **regular synchronisation between microcontrollers:** the two microcontrollers are expected to synchronise every 100ms maximum by checking the signature (CRC) of their memory spaces. Executing a loop with for example 100 millions steps would similarly trigger the panic mode.

It is also possible to reach the RAM limit by allocating large tables (containing 48700 *uint8_t* for example) or change the board id (jumper) during program execution to respectively prevent or stop its execution.

7 Dissemination

A first starter kit, SK_0, containing the IDE and the execution platform, was released by the end of 2017[5], presented and experimented at the occasion of several hands-on sessions organized at university sites in Europe, North and South America. Audience was diverse, ranging from automation to embedded systems, mechatronics, computer science and formal methods. Results obtained are very encouraging:

- Teaching formal methods is eased as students are able to see their model running in and interacting with the physical world. It was the occasion to demonstrate how formal methods could be used with embedded systems and IoT. Fruitful discussions took place about how to specify/guaranty performances, what can or cannot be proved with such systems, etc.

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

- Less theoretic student profiles (computer science, mechatronics, automation) may be introduced/educated to more abstract aspects of computation. *clock* and *combinatorial* exercises were a starting point for specification enrichment and the discovery of the formal proof. Of course, the pedagogical objective in term of formalization was lower than with more formal profiles, but the students managed to understand the absence of programming error and the non-deterministic substitutions for simple modelling.
- The platform has demonstrated a certain robustness during all these manipulations and has been enriched with the feedback collected so far. Several electronics/software errors were detected during the preparation of the course when designing exercises, others during these exercises.
- The IDE GUI was improved with the automation of the code generation process and the display of a carousel showing graphically the progress of the generation. The configuration of the board was also simplified, by displaying the position of the switches on the board and by filling the configuration file with default input and output names.
- CLEARSY Safety Platform is used to teach in Master 2[6] in universities and engineering schools. Electronic documentation[7] is used to structure the courses and is updated every 2 months. With 3 inputs and 2 inputs, the starter kit SK0 is for discovering the technology; another version of the board is planned for 2020 able to handle more I/O (up to 64).

8 Ready for Industry

The SK_0 board provides a good introduction to the programming of safety critical systems. However the framework proposed is mainly aimed at education and not perfectly fit for industry:

- the number of I/O is reduced (5)
- the programming schema is simple: read inputs, compute, set outputs. Identical algorithms are executed by I_1 and I_2.

The CLEARSY Safety Computer 0 (abbreviated as CS_0) (Fig. 12) was designed to offer more flexibility by providing:

- more I/O. The associated mother board brings 32 inputs and 32 outputs, all digital.
- and a programming model less constrained:
  - Safety functions are still programmed and proved in B, but are callable individually from the C program, in the main loop or associated to an interrupt vector.
  - Mandatory (watchdog-based) safety verification are still performed by the safety library but the developer is now responsible for calling in time the verification functions that keep the watchdogs alive.
  - Computation could be asymmetric between I_1 and I_2.

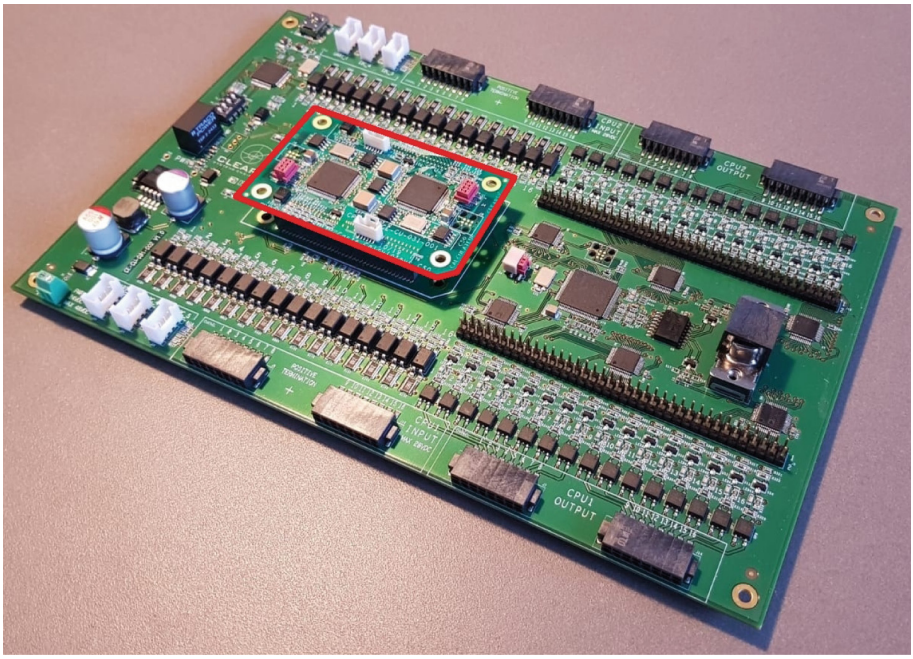

Fig. 12. The CS0 daughter board safety computer on the left, plugged on the mother board.

The CS_0 only embeds the 2 microcontrollers on a smart card format daughter board while the I/O and the power supply are located on a hosting mother board.

9 Conclusion and Perspectives

The CSSP provides a new way of practising formal methods by allowing students/engineers to connect formal models with the surrounding world. The CSSP is also used to create safety-critical systems, able to be certified at the highest safety levels[8,9,10].

As a consequence, the B0 modelling language has been (even more) restricted to allow an easier certification because of the simplicity of the tool chain. These restrictions oblige to have more verbose models (with more lines and more nesting levels). Even if these constraints could be released/removed in the future, the obtained proof automation level is a real improvement that would certainly ease its adoption in engineering processes. The invention of the CLEARSY Safety

[6] Second year of a Master's degree.
[7] Available at https://github.com/CLEARSY/CSSP-Programming-Handbook.
[8] Generic product certificate, CERTIFER 8891/200-1, 27th Feb 2017 SIL4.
[9] System certificate BUREAU VERITAS 6393741 3rd March 2017 SIL3.
[10] Generic product certificate BUREAU VERITAS 7092509, 23rd July 2019 SIL4.

Platform also paves the way for a broader use of the B formal method, in the railways and in other safety-related domains like energy or autonomous vehicles.

Acknowledgements. The work and results described in this article were partly funded by BPI-France (Banque Publique d'Investissement) and Métropole Aix-Marseille as part of the project LCHIP (Low Cost High Integrity Platform) selected for the call AAP-21.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Boulanger, J.: Formal Methods: Industrial Use from Model to the Code. Wiley, Hoboken (2013)
3. Gashi, I., Povyakalo, A., Strigini, L.: Diversity, safety and security in embedded systems: modelling adversary effort and supply chain risks. In: Proceedings of 2016 12th European Dependable Computing Conference (EDCC), Gothenburg, pp. 13–24 (2016)
4. Lecomte, T.: Safe and reliable metro platform screen doors control/command systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 430–434. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_32
5. Lecomte, T.: Applying a formal method in industry: a 15-year trajectory. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 26–34. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_3
6. Lecomte, T.: Double cœur et preuve formelle pour automatismes sil4. 8E-Modèles formels/preuves formelles-sûreté du logiciel (2016)
7. Lecomte, T., Deharbe, D., Prun, E., Mottin, E.: Applying a formal method in industry: a 25-year trajectory. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 70–87. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_6
8. Sabatier, D.: Using formal proof and B method at system level for industrial projects. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) RSSRail 2016. LNCS, vol. 9707, pp. 20–31. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33951-1_2