# The CLEARSY Safety Platform: 5 Years of Research, Development and Deployment

Thierry Lecomte[a], David Deharbe[a], Paulin Fournier[a], Marcel Oliveira[b]

[a]*CLEARSY, Aix en Provence, France*
[b]*IMD, Universidade Federal do Rio Grande do Norte, Brazil*

**Abstract**

The CLEARSY Safety Platform (CSSP) was designed to ease the development of safety critical systems and to reduce the overall costs (development, deployment, and certification) under the pressure of the worldwide market. A smart combination of hardware features (double processor) and formal method (B method and code generators) was used to produce a SIL4-ready platform where safety principles are built-in and cannot be altered by the developer. Summarizing a 5-year return of experience in the effective application in the railways, this article explains how this approach is a game-changer and tries to anticipate the future of this platform for safety critical systems. In particular, the education of future engineers and the seamless integration in existing engineering processes with the support of Domain Specific Languages are key topics for a successful deployment in other domains. DSL like Robosim to program mobile robots and relay circuits to design railway signalling systems are connected to the platform.

*Keywords:* formal methods, safety critical, software development, railway

## 1. Introduction

In several industrial standards (EN50128 for SIL3/SIL4, IEC61508 SIL3/SIL4, ISO 26262 for ASIL4), formal methods are highly recommended when developing safety critical software for the highest safety levels, for the specification, the development and/or the verification phases. 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 [11][12][19][14], more easily certifiable, a generic, safe execution platform has been researched for years, combining safety electronics and defect-free proven software[1]. 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*[13] 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[2], 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). As the safety demonstration does not require any specific feature for the input B model, it could be handwritten

*Email addresses:* `thierry.lecomte@clearsy.com` (Thierry Lecomte), `david.deharbe@clearsy.com` (David Deharbe), `paulin.fournier@clearsy.com` (Paulin Fournier), `marcel@dimap.ufrn.br` (Marcel Oliveira)

[1]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.

[2]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 expected to improve automatic proof.

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 demonstrates how redundant hardware and formal method can be combined to obtain a platform able to execute a safety critical application, while the developer only has to focus on the functional aspect. Hence software development may be delegated to non-expert engineers and testing is limited to validation, unit and integration testing being replaced by mathematical proof. The main contribution is cost reduction for application development, certification and deployment, as the execution platform is an order of magnitude cheaper than existing off-the-shelf solutions. It would also enable the embedding of new safety-related devices that were not previously considered because of their expensiveness.

This paper is structured in six parts. Section 2 introduces the Terminology. Section 3 provides the rationale for designing the CLEARSY Safety Platform. Section 4 briefly introduces the B method. Section 5 introduces the architecture and safety principles of the CLEARSY Safety Platform showing how the combination of the B formal method and electronic diversity ensures a high safety level, and demonstrating how the process is automated and how high profile engineers are not required anymore to complete the software development. Section 6 details the on-going connection with Domain Specific Languages in order to ease the CLEARSY Safety Platform adoption by non-formalists. More abstract, section 7 shows how an Event-B model of a system could be used to derive a CLEARSY Safety Platform program, allowing to increase the level of confidence on the software specification.

## 2. Terminology

This section contains specific definitions, concepts, and abbreviations used throughout this paper.

**Atelier B** is an Integrated development environment (IDE) supporting the B method and the B language for software development, and Event-B for system-level analysis. Atelier CSSP is Atelier B extended with redundant code generator toolchain, bootloader, and a new project type (CSSP project).

**B0** is a subset of the B language 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.

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

**CRC** put for cyclic redundancy check[20], is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

**Diversity** refers to a method for improving the reliability of a message signal by using two or more communication channels. In our case, two diverse code generators producing different binaries from a single model allow to detect compilation errors during execution by comparing their behaviour.

**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[21] that conveys binary information in ASCII text form. It is commonly used for programming microcontrollers, EPROMs, and other types of programmable logic devices.

**PLC** put for programmable logic controller[23], 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.

**Ladder logic** is a programming language[22] that represents a program by a graphical diagram based on the circuit diagrams of relay logic hardware.

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

**Safety belt** refers to the safety properties that are part of the modelling. This modelling results have to be proved against these properties.

**Safety computer** usually refers to a computer controlling a system where the emission of an erroneous output could injure or kill people. Safety techniques (error detection, redundancy, etc.) have to be used to lower the probability of occurrence of such a failure remains below an acceptable level defined by standards.

**SIL** put for Safety Integrity Level[24], 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.

**Output states (memory vs physical)** A controller computes new values for its outputs every cycle. These values (stored in memory) are used to change (with signal conversion components) the physical state of the outputs. Identity between values in memory and outputs physical state are checked regularly (with a monitor checking the signal conversion) to assess if the controller is still able to control. Depending on how the outputs are implemented (relays, transistors), changing state may take more or less time and identity check has to be delayed accordingly.

## 3. Rationale

Developing a safety computer[5] from scratch is not something you easily decide because of the effort required to obtain such a device (several millions of Euros have been spent to develop the current CLEARSY Safety Platform). Two kinds of devices are currently available on the market for safety critical applications: PLCs (Siemens[3], HIMA[4], etc.) and SIL3/SIL4-ready boards (MEN[5], SC3[6] etc.). 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.

Our first safety system development [15] was aimed at controlling platform screen doors with a Siemens S7 PLC. The PLC was programmed with Ladder Logic[7], one of the five programming languages mentioned by the railway standards. This PLC comes with a SIL3 certificate requiring that Ladder Logic programs must be entered by using their own internal editor. At that time, the program specification was initially written in B (and somehow loosely coupled with a preliminary system level modelling in Event-B), then implemented in B0. The B0 program was proved to comply with its specification, then translated into Ladder Logic. The resulting Ladder Logic program was manually typed in the Siemens S7 PLC. As the B0-to-ladder translator was not formally developed / verified, peer reviews took place to check the conformance of the code in the PLC with the B0 program (the code generated was very close to the B0). Because of these human activities, the development process was quite heavy and the use of formal modelling and proof was mostly nullified by these systematic reviews (many iterations were necessary to fine tune the software). Later, unfortunately (or fortunately if we consider what happened in reaction), one of our competitors managed to literally copy our system (hardware and software) and to sell it at a lower price. We then decided to develop our own solution based on the combination of redundant hardware and proven software developed with B. Producing our 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.

Finally, the CSSP is a game changer as it proposes an alternative to the two existing safety PLCs / safety-ready boards (we are not considering in-house PLCs developed by train manufacturers which are not commercially available):

[3]https://new.siemens.com/global/en/products/automation/systems/industrial/plc.html
[4]https://www.hima.com/en/industries-solutions/overview-of-all-hima-solutions
[5]https://www.duagon.com/products/computing/safe-computing-systems
[6]https://sc3automation.com/

- black-box PLCs with a SIL3-SIL4 stamp but also a very constrained development framework and the inability to adapt to specific requirements (signal processing on the input signal for example). The safety demonstration heavily relies on the PLC safety certificate and the bounded development environment.
- safety-ready boards, providing a kit of features that the developer needs to connect and activate cautiously, in time and in order, together with the application specific code. The safety demonstration has to be fully designed by the developer.

The third way proposed by the CSSP offers the freedom to tailor the application (input/output signal processing, variable cycle time, asymmetric computing among the two processors, etc.) and to ease the safety demonstration, if the application complies with safety hypotheses coming with the certification kit. These hypotheses, named "safety application conditions", not described in this paper, have been identified during previous certification processes and are sufficient to complete the safety demonstration. Having the application B project fully proved is one of them.

## 4. Introduction to the B Method

B[1] is a method for specifying, designing, and coding software systems. It covers central aspects of the software life cycle (Fig. 1): the writing of the technical specification, the design by successive refinement steps and model decomposition (layered architecture), and the source code generation.

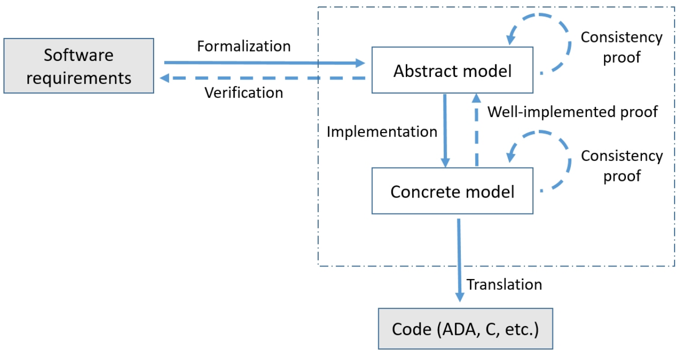


Figure 1: A typical B development cycle, from requirements to code.

B is also a modelling language that is used for both specification, refinement (Fig. 2), and implementation (Fig. 3). It relies on substitution calculus, first order logic and set theory. All modelling activities are covered by mathematical proof that finally ensures that the software system is correct.

B is structured with modules and refinements. A module is used to break down a large software into smaller parts. A module has a specification (called a machine) where are formalized both a static and a dynamic description of the requirements. It defines a mathematical model of the subsystem concerned:

- an abstract description of its state space and possible initial states,
- an abstract description of operations to query or modify the state.

This model establishes the external interface for that module: every implementation will conform to this specification. Conformance is assured by proof during the formal development process. A module specification is refined. It is re-expressed with more information: adding some requirements, refining abstract notions with more concrete notions, getting to implementable code level. Data refinement consists in introducing new variables to represent the state variables for the refined component, with their linking invariant. Algorithmic refinement consists in transforming the operations for the refined component. A refinement may also be refined. The final refinement of a refinement column is called the implementation, it contains

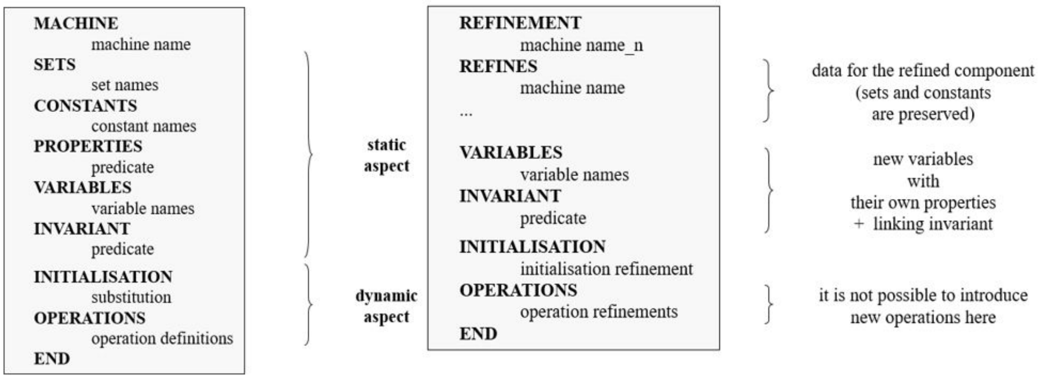


Figure 2: Structure of MACHINE and REFINEMENT components.

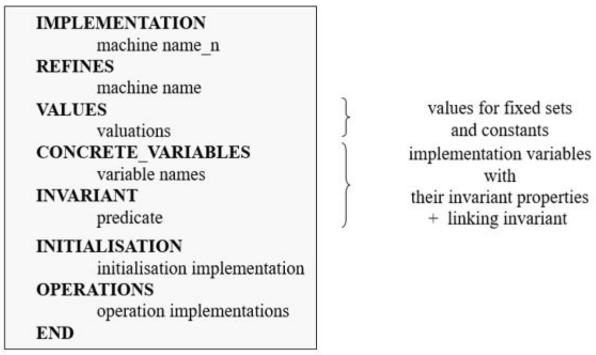

IMPLEMENTATION
machine name_n
REFINES
machine name
VALUES
valuations
CONCRETE_VARIABLES
variable names
INVARIANT
predicate
INITIALISATION
initialisation implementation
OPERATIONS
operation implementations
END
values for fixed sets and constants
implementation variables with their invariant properties + linking invariant

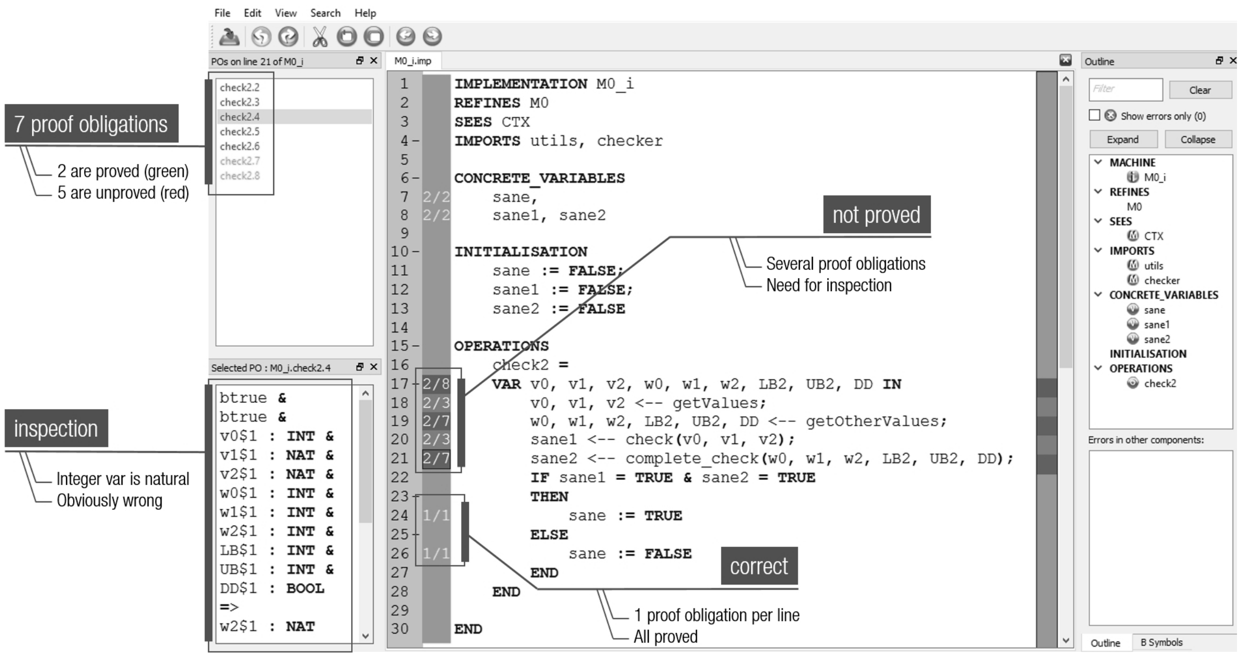
Figure 3: Structure of IMPLEMENTATION component.

only B0-compliant models. In a component (machine, refinement, or implementation), sets, constants, and variables define the state space while the invariants define the static properties for its state variables. The initialisation phase (for the state variables) and the operations (for querying or modifying the state) define the way variables are modified. From these, proof obligations are computed such as: the static properties are consistent, they are established by the initialisation, and they are preserved by all the operations. Atelier B contains a model editor merging model and proof (Fig. 4) by displaying the number of proof obligations associated to any line of a B model, its current proof status (fully proved or not) and the body of the related proof obligations.

Finally a B project is a set of linked B modules. Each module is formed of components: an abstract machine (its specification), possibly some refinements and an implementation. The principal dependencies links between modules are IMPORTS links (forming a modular decomposition tree) and SEES links (read only transversal visibility). Sub-projects may be grouped into libraries. A software developed in B may integrate or may be integrated with traditionally developed code.

## 5. Architecture and Safety Principles

This section contains a description of the platform architecture and its programming model, as well as a summary of the safety principles. The safety case contains all the details leading to complete demonstration (SIL4) but are not disclosed here (the safety case is around 120 pages). The CSSP has already been certified 3 times.



```
check2.2
check2.3
check2.4
check2.5
check2.6
check2.7
check2.8
```

```
btrue &
btrue &
v0$1 : INT &
v1$1 : NAT &
v2$1 : NAT &
w0$1 : INT &
w1$1 : INT &
w2$1 : INT &
LB$1 : INT &
UB$1 : INT &
DD$1 : BOOL
=>
w2$1 : NAT
```

```
IMPLEMENTATION M0_i
REFINES M0
SEES CTX
IMPORTS utils, checker

CONCRETE_VARIABLES
    sane,
    sane1, sane2

INITIALISATION
    sane  := FALSE;
    sane1 := FALSE;
    sane2 := FALSE

OPERATIONS
    check2 =
    VAR v0, v1, v2, w0, w1, w2, LB2, UB2, DD IN
        v0, v1, v2 <-- getValues;
        w0, w1, w2, LB2, UB2, DD <-- getOtherValues;
        sane1 <-- check(v0, v1, v2);
        sane2 <-- complete_check(w0, w1, w2, LB2, UB2, DD);
        IF sane1 = TRUE & sane2 = TRUE
        THEN
            sane := TRUE
        ELSE
            sane := FALSE
        END
    END
END
```

Figure 4: Atelier B model editor showing proof status.

## 5.1. *Introduction*

The CLEARSY Safety Platform is a generic PLC able to perform command and control over inputs and outputs. For safety critical applications, the PLC has to be able to determine whether it is fully functional or not. In case of failure, the PLC should move to restrictive mode where all the outputs are deactivated. The stronger the risk of harming people in case of failure, the higher the Safety Integrity Level. For SIL3 and SIL4, the computations have to be performed by a minimum of two processors and checked with a voting system. The verification listed below in Table 1 is used to detect PLC failures and to trigger a move to restrictive mode.

## 5.2. *Architecture*

The CLEARSY Safety Platform is made of two parts: an IDE to develop the software and an electronic board to execute this software.

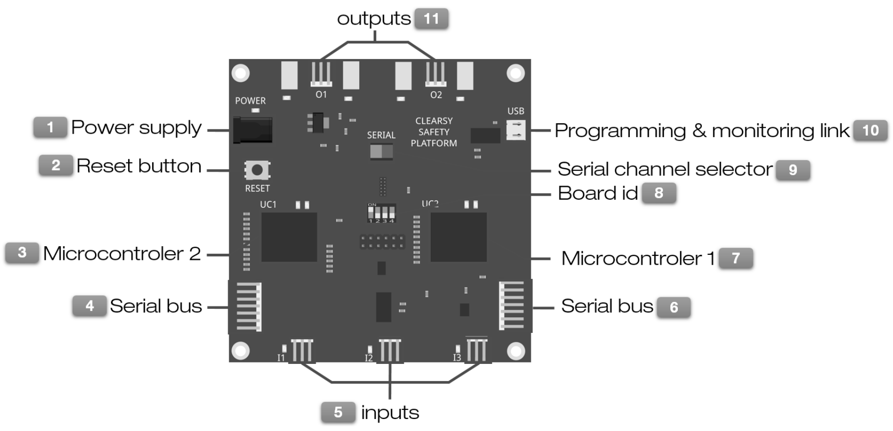


Figure 5: The CLEARSY Safety Platform starter kit $SK_0$ board

The overall architecture (Fig. 5) is common to all instances of the CLEARSY Safety Platform (starter kits $SK_0$ and $SK_1$). The differences lie in the number of digital (binary) IOs: 5 for $SK_0$, 28 for $SK_1$. Future instances of the CLEARSY Safety Platform will feature analog IOs and networking services (messaging) through a maintenance processor i.e. a non safety-related processor in charge of spying the microcontrollers bus and to emit traces of execution to the outside. From a safety point of view, the current architecture is valid for any kind of mono-core processor. The decision of using PIC32 microcontrollers (able to deliver around 50 DMIPS) was made based on our knowledge and experience of this processor. Implementing the CLEARSY Safety Platform on other hardware (STM32 for example) would "only" require to modify the existing electronic board and software tools, without much impact on the safety demonstration. Note that the PIC32 microcontrollers used by the hardware platform are commercial products. From the standards, their reliability is considered as $10^{-5}/h$.

The full process is described in Fig. 6 where rounded boxes are tools and rectangles are files; $\mu C_1$ and $\mu C_2$ are PIC32 microcontrollers.

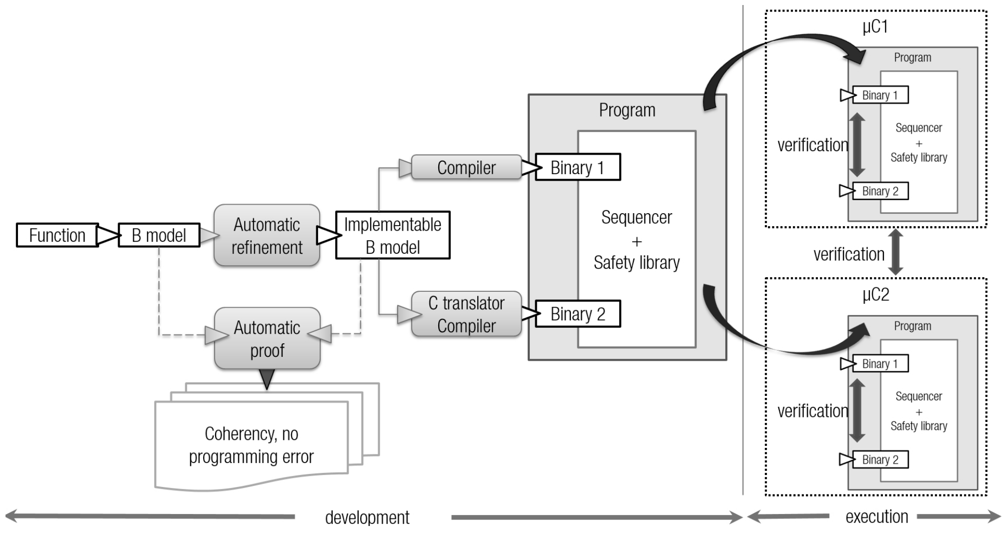


Figure 6: Full path from function description to safe execution

The CLEARSY Safety Platform development cycle strictly follows the B method which can be summarized as:

- specification model is written first from the natural language requirements (Function), then comes the implementation model, both using the same language (B). The implementable B model could be automatically refined with Atelier B BART tool, but it requires to have a fully deterministic model.
- models are proved to be coherent and to be correct refinements. The proof is automatic if the complexity of the model is not too high for the Atelier B theorem prover. Frequent interactive demonstrations can be turned in proof tactics to be applied automatically,
- source code or binary is generated from implementation model:
  - Binary 1 (HEX file) is directly compiled from the implementation B model. The compiler has been developed in-house for supporting this technology.
  - Binary 2 (HEX file as well) is generated in two steps. First, Implementation models are translated to C, using the Atelier B C code generator. Then the C code is compiled with gcc.
- The two binaries are linked to a top-level sequencer and a safety library, both software developed in B by the CSSP IDE development team once for all, to constitute the final software.

- This software is then loaded on the flash memory of the two microcontrollers (bootload mode).
- When the board enters the execution mode or is reset, the content of the flash memory is copied in RAM for both microcontrollers which start executing it.
- For each microcontroller, the top-level sequencer enters a never-ending loop and
  - calls in sequence Binary 1 then Binary 2 for one iteration
  - calls the safety library in charge of performing verification.
  - If the verification fails, the board enters panic mode, deactivates its outputs and enters an infinite loop doing nothing.

### *5.3. Programming*

The process starts with the specification of the function to develop commonly expressed with natural language. The developer has to provide a B model of it (specification and implementation) matching the following pattern:

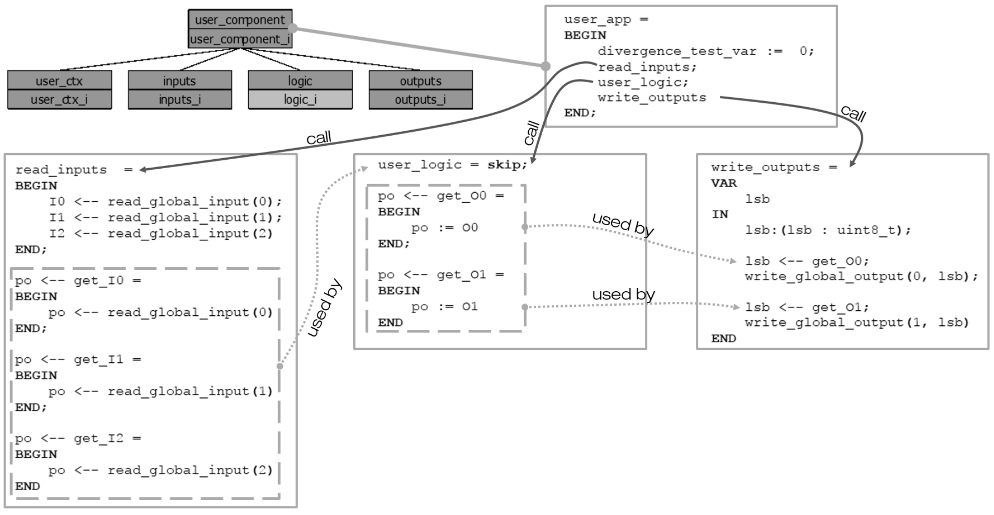
- The function to program is a loop, where the following steps are performed repeatedly in sequence:
  - the inputs are read. Inputs are the same for $\mu C_1$ and $\mu C_2$, unless inputs are captured at different times, in which case the different values would cause the platform to enter panic mode.;
  - some computation is performed in relation with the inputs/outputs status, local variables and the time elapsed since the last reset;
  - the outputs are set.
- The steps related to inputs and outputs are fixed and cannot be modified.
- Only the computation may be modified to obtain the desired behaviour.

The Atelier CSSP creates a skeleton of a B project. Fig. 7 shows the behavioural part of the project. The top level module is made of two components: *user_component* (the specification) and *user_component_i* (the implementation). This implementation imports 4 modules: *user_ctx*, *inputs*, *logic*, and *outputs*. *user_ctx* contains only constants and sets defined by the developer; *inputs* and *outputs* contain variables and operations for accessing the inputs (read access) and outputs (write access). *logic* contains the control & command logic of the board. *user_ctx* and *logic* are the only two modules to modify. Other modules could be required by *logic* and have to be imported by *logic_i*. Programming the board consists of:

- describing, in the component *logic*, the behaviour of the board;
- declaring and using constants, defined in the component *user_ctx*;
- reusing operations defined in the component *inputs* and in the safety library.

The B language supported by the CLEARSY Safety Platform differs from the one described in [1], among which (Fig. 8):

- safety variables are all unsigned integers, coded either on 8, 16, or 32 bits.
- digital inputs and outputs are unsigned integers coded on 8 bits (uint8_t). Their values are either IO_OFF or IO_ON.
- testing conditions are restricted to one term only. Multiple conditions have to be in nested IF.
- local variables have to be typed with a "becomes such as" substitution.
- arithmetic operators able to produce overflow (+, -, *) are replaced by non-overflowing operators using modulo calculation to produce a result in the range of the variable receiving the result.



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

```
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
```

Figure 7: A typical CLEARSY Safety Platform project. The divergence_test_var is exposed as it is required by the code generation toolchain but has no impact on the modelling.

All the operations defined in the components *inputs* and *outputs* have to remain unchanged, as well as the accessing functions get_* defined in the component *logic*. Although the implementable B model, or *implementation*, is usually developed manually, it may also be automatically generated with the B Automatic Refinement Tool. The B models are proved (mostly automatically as the level of abstraction of typical command & control applications is low) to be coherent and to contain no programming error. From the implementable model, two binaries are generated:

- Binary$_1$, obtained via a dedicated compiler (developed by CLEARSY) transforming a B model into a HEX file,
- Binary$_2$, produced with the Atelier B C code generator, then compiled with the GCC compiler into another HEX file.

Each binary represents the same function but is supposed to be made of different sequences of instructions because of the diversity of the tool chains. Then the two binaries Binary$_1$ and Binary$_2$ are linked with:

- a sequencer, in charge of 1) reading inputs, 2) executing once Binary$_1$ then Binary$_2$, 3) setting the outputs;
- a safety library, in charge of performing safety verification. In case verification fails, the board enters panic mode, meaning the outputs are deactivated (no power is provided to the Normally Open (NO) outputs, so the output electric circuits are open), the board status LED starts flashing, and the board enters an infinite loop doing nothing. A hard reset (power off or reset button) is the only possibility to interrupt this panic mode.

The final program is thus made of Binary$_1$, Binary$_2$, the sequencer and the safety library. The memory mappings of Binary$_1$ and Binary$_2$ are separate. This program is then uploaded on the two micro-controllers $\mu C_1$ and $\mu C_2$.

## *5.4. Safety Principles*

For the safety case, the feared event is the wrong powering of one of the outputs i.e. this output has to be OFF (the relay should not be powered) but it is currently ON (the relay is powered). The power is provided by both microcontrollers, so if one of the two is reset, it would not power the relay and the board is in a restrictive safe state. The safety principles are distributed on the board and on the safety library. The

```
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 := IO_OFF;
        IF i1_ = IO_ON THEN
            IF i2_ = IO_ON THEN
                IF i3_ = IO_ON THEN
                    O1 := IO_ON
                END
            END
        END;
        IF O1 = IO_ON THEN
            O2 := IO_OFF
        ELSE
            O2 := IO_ON
        END
    END
END
```

Figure 8: One implementation of the operation user_logic

safety case demonstrates that the verification performed during development and execution are sufficient to ensure the target safety integrity level.

The bootloader, on the electronic board, checks the integrity of the program (CRC, separate memory spaces). Then both microcontrollers start to execute the program. During execution, the following verifications are conducted. If any of these verification fails, the board enters the panic mode:

- internal verification (performed within a single microcontroller):
  - every cycle, Binary$_1$ and Binary$_2$ data memory spaces (variables) are compared within each microcontroller;
  - regularly, Binary$_1$ and Binary$_2$ program memory spaces are compared. This verification is performed "in the background" over thousands / millions cycles - to keep a reasonable cycle time.;
  - regularly, the identity between memory outputs states and physical output states is checked to detect if the board is unable to command the outputs.
- external verification (performed between both microcontrollers):
  - regularly (every 50ms at the latest), data memory spaces (variables) are compared between $\mu C_1$ and $\mu C_2$.

The whole process is fully supported by dedicated tools. None of the tools part of the toolchain are proved to be correct. In Fig. 9, the tools and generated text and binary files are made explicit for both the application (the process is conducted every time an application is developed) and the safety belt (developed once for all by the IDE development team [7]). All the tools come from Atelier B, except:

- the B to HEX compiler, initially developed to control platform screen doors for metro lines in Brazil. This tool proceeds in two steps: a translation from B to ASM MIPS, then from ASM MIPS to HEX. In order to ease debugging as ASM MIPS to HEX is a straightforward line-to-line translation.

[7]Note that from the abstract formal model, one part of the software is developed in B with a concrete formal model, while the other part is developed manually. It happens when using B provides no added-value (for example low-level IO). A component modelled in B and implemented manually is called a basic machine.

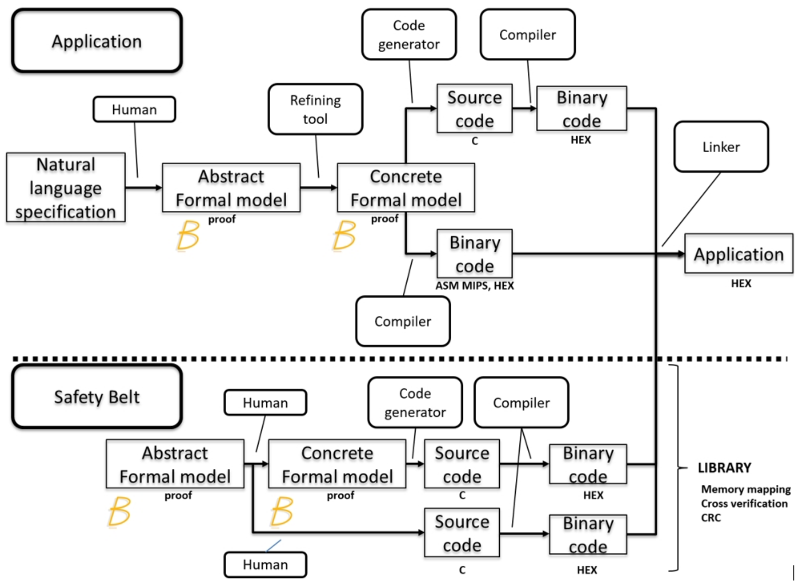


Figure 9: Tools and files involved in the generation of the software

- the C-to-HEX gcc compiler.
- the linker that combines the two HEX files with the safety sequencer and libraries.
- the bootloader.

Some of the tools have been "certified by usage" since 1998 [4], but the newest tools of this toolchain 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, the development and verification process, and other elements). 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 four instances of the function and the two microcontrollers. This way, similar bugs that could affect two independent tools at the same time and with the same effects are simply neglected: the standards incorporate the assumption that two tools developed with independent teams using different technologies could not show exactly the same buggy behaviour[8]. So a bug will always be detected by comparing the behaviour of the two tools.

The safety is built on top of several principles:

- a B formal model of the function to develop, proved to be coherent, to correctly implement its specification, and to be programming error-free i.e. no division by zero, no overflow, no access to a table outside of its range;
- four instances of the same function running on two micro-controllers (two per micro-controller with different binaries obtained from diverse tool-chains) and the detection of any divergent behaviour among the four instances;
- the deferred cross-verification of the programs on-board the two micro-controllers;
- outputs require both $\mu C_1$ and $\mu C_2$ to be live and running as one provides energy and the other one the command;

[8]Common cause failures may happen with shared conditions (same compiler, same library, same programming language, same design, same team, etc.) and break the diversity principle. If the absence of common cause failures is established during the safety analysis then the probability of occurrence of the same failure on the two paths at the same time is considered to be lower than the probability of disappearance of the atmosphere - hence neglected in the safety case.

Table 1: Verification performed during development and execution

| **Stage** | **#** | **Failure** | **CSSP verification** |
|---|---|---|---|
| specification | 1 | Typing error | Typechecker tool detects typing error |
| specification | 2 | Specified behaviour incompatible with invariant properties | Unprovable proof obligation indicates specification mistake |
| implementation | 3 | Typing error | Typechecker tool detects typing error |
| implementation | 4 | Implemented behaviour incompatible with invariant properties | Unprovable proof obligation indicates implementation mistake |
| implementation | 5 | Implemented behaviour incompatible with specified behaviour | Unprovable proof obligation indicate implementation mistake |
| implementation | 6 | Overflow capable arithmetic operators used instead of dedicated ones | Detected by the B-to-HEX compiler |
| implementation | 7 | IF clause with more than one condition (B0 language restriction) | Detected by the B-to-HEX compiler |
| implementation | 8 | LOCAL variables not typed before use (B0 language restriction) | Detected by the B-to-HEX compiler |
| code generation | 9 | Syntax errors in the C generated code | Detected by the MICROCHIP compiler |
| code generation | 10 | Incorrect naming in the C generated code | Detected by the linker |
| code generation | 11 | Incorrect memory map | Memory overlap detected by the bootloader |
| compilation | 12 | Wrong binary code generated | Detected during execution by the safety library by comparing binary$_1$ and binary$_2$ variables in memory with CRC on the same $\mu C$ |
| uploading | 13 | Incorrect transfer between host and electronic board | Detected by bootloader during upload (CRC) and during execution over several cycles |
| execution | 14 | RAM error (variables) | Detected by comparing binary$_1$ and binary$_2$ variables in memory with CRC on the same $\mu C$ |
| execution | 15 | RAM error (program) | Detected by comparing binary$_1$ and binary$_2$ program in memory with CRC with the other $\mu C$ |
| execution | 16 | Failure of one $\mu C$ | Detected by handshake between $\mu C_1$ and $\mu C_2$ at least every 50 ms |
| execution | 17 | Outputs not command-able | Detected by checking physical state and command issued by the software |

- physical output states are regularly verified to comply with the software output states, to check the ability of the board to command its outputs;
- input signals are continuous (0 or 5V) and are made dynamic (addition of a frequency signal) in order not to consider short-circuit current as high level (permissive) logic.

The verification performed by the CLEARSY Safety Platform, either during development or execution stages, is summarized in Table 1.

The safety critical electronic board needs some vital elements to comply with the highest SIL requirements, such as:

- ensuring galvanic isolation between the two half-boards, to avoid that one side of the board wrongly provides energy to the other side's outputs
- activate safety outputs with a sinusoidal signal instead of a continuous signal, to ignore fault current. The micro-controller needs to be alive to generate the sinusoidal signal. An electrical transformer connected to the output line will generate power only if powered by alternative current.

These features, that are not implemented on the starter kits, are only needed for real-life safety critical systems and do not prevent developers, whether students, researchers or engineers, to get educated with the CLEARSY Safety Platform and to develop prototypes.

## 6. Engineering through DSLs

The CLEARSY Safety Platform is a software plant where a B model is automatically proved and transformed into a binary program that executes safely on dedicated hardware. The connection between Domain Specific Languages and the CLEARSY Safety Platform, through a translation from DSL to B, would entitle engineers to make profit of the CLEARSY Safety Platform safety features and to quite easily obtain a safety function issued from the used usual modelling language. Two experiments are being conducted: one with relay schemes for the French railways to replace wired-logic devices by programmed ones, the other with RoboSim to address the robotics domain. With these two case-studies, the objective is not to directly obtain a safety critical design but to demonstrate that the translation from these two formalisms to B (as supported by the CSSP) is doable and enables a path from one DSL model to a safe execution. In both cases, the properties expressed in the B specification are almost minimal as they are expected to have been verified at the DSL level.

### *6.1. Relay Circuits*

Relay circuits are electrical circuits that have been first considered for translation and support by the CSSP.

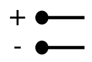
#### *6.1.1. Technical and Industrial Context*

Despite the existence of digital systems, it is still common to use relay-based systems. Indeed, in the railway signaling domain, the interlocking systems responsible for allowing or denying trains movement are still nowadays implemented by electrical circuits containing relays. These electrical circuits receive, use and transmit information through the use of electromagnetic switching elements. These elements are composed of electromagnet (coils) and contacts. The natural state of the contact (dictated by gravity) is modified by the state of the coil. When the coils are activated, the state of the contacts is modified thus powering or disabling other coils. Such combinations are a mean to implement logic functions.

Such electrical systems are designed by drawing electrical circuits, namely, relay circuits (we refer the interested reader to *e.g.* [18]). An example of such a specification is presented in Fig. 10. It is a circuit for controlling the state of a color light signal installed sideways of the tracks. This design has been provided to us by *SNCF Réseau* (owner and main manager of the French railway network).

These relay circuits are mainly composed of :

- Electrical sources (positive and negative):



- Inputs, which are a special case of electrical sources that can be either activated or disabled.
- Outputs, which are just electrical sources but that have effect outside of the system and thus are monitored.
- Mono-stable relays, which are composed of a single electromagnet; they are active when an electric current goes through them, inactive otherwise.

  In the example, the relays are depicted from left to right: CM, CFR, RPCS, CRR, CA, RPA, BS, EX1 and EX2. They are pictured with a rectangular shape containing a chevron.
- Contacts, which have two states: open or closed. A closed contact lets the current flow through whereas an open contact prevents it. Contacts are associated to a relay. There are two types of contacts:

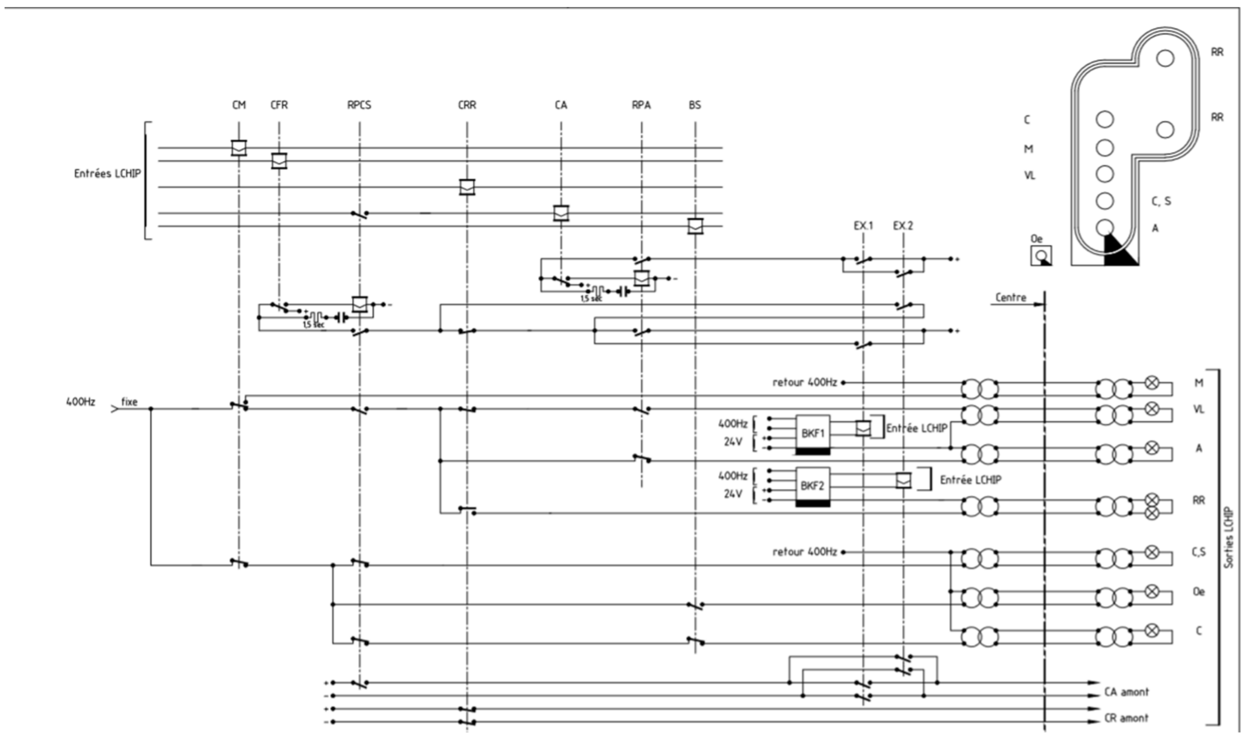


Figure 10: Example of a relay circuit.

- a *normally opened* contact requires its relay to be active to close;
- a *normally closed* contact requires its relay to be active to open.

In the example, the contacts are aligned vertically with their associated relay and are visually connected with a dashed line. For instance, there are two contacts associated with the relay BS. The visual position of the contact indicates whether it is normally opened (drawn below the wire) or normally closed (drawn above the wire). For example, the top-most contact associated to BS is normally opened while the bottom-most is normally closed.

Notice also that, in this drawing, relay CRR is active since its normally-opened contacts are closed and, conversely, its normally-closed contacts are open. All the other relays are inactive.

Note that other components exist, such as light control block (BKF), timers, bi-stable relays and timed relays.

### *6.1.2. Translating Relay Circuits Design to B Components*

To re-use the design of relay circuits to implement a safe digital circuit, we developed a prototype allowing to translate relay circuits to B components. This prototype is composed of the following two steps:

1. Enter the circuit structure using a simple interface based on highlighting.
2. Translate automatically the entered circuit to B.

These steps are detailed in the following. Also, since each such step may contain errors, and since this system is critical (a wrong output may induce the train driver to enter a track section allocated to another train), it is necessary to verify their output. We also present how these steps may be verified.

*Highlighting.* First the user uses a graphical user interface that allows to highlight the circuit piece-wise. The principle is that the user must highlight every single linear wire (ending either in inputs, outputs, electrical sources or junctions), and click sequentially on every element on the wire. Every time the user clicks on an element, the interface queries the nature and the identity of the component. This process allows to convert the graphical scheme to a simple tabular format.

Highlighting provides the user with a visual clue of which part of the circuit have already been entered. Once the user has completed entering all the relevant parts of the circuit, the first step is finished.

A visual representation of the result of highlighting the scheme of Fig. 10 is given in Fig. 11. Parts of the circuit that have not been entered are greyed out. In the final setup, these parts will not be executed by the CLEARSY Safety Platform, and they shall be implemented by other physical components.

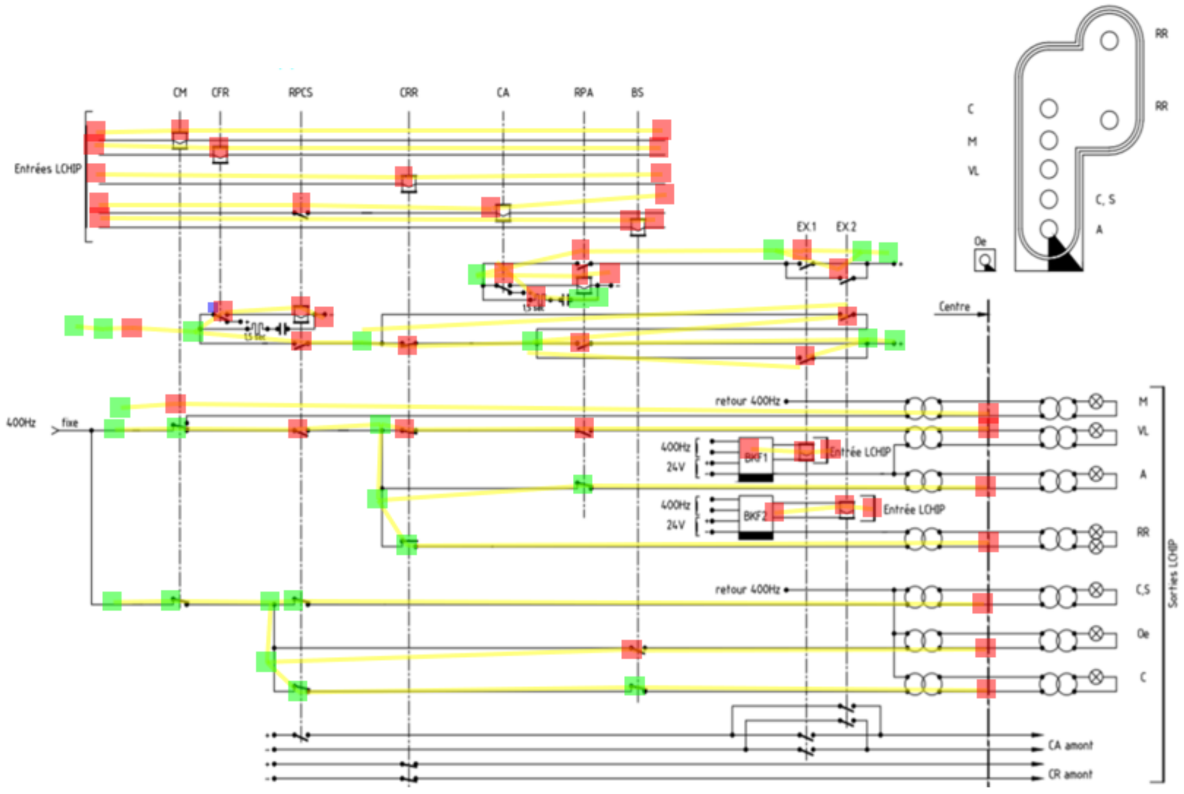


Figure 11: Highlighting of the scheme of Fig. 10. Highlighted wires appear in yellow, entered circuit elements appear in green (when energized in default state) and red (otherwise) squares

*Translation to B.* Once the relevant part of the circuit has been entered by highlighting, tables are filled with all the information required to perform the translation into B components that are compatible with the CLEARSY Safety Platform. The principles of the translation are the following.

The inputs of the CLEARSY safety platform are the inputs of the relay circuit, i.e., electrical sources identified as inputs by the user. Likewise, the output of the CLEARSY safety platform are the outputs of the relay circuit.

The variables encode the states of the relays (either active or inactive). For each relay, there is a corresponding bi-valuated variable. The computation done during a cycle of the Clearsy safety platform consists of computing the new state of the relays according to the state of the inputs and the previous states of the relays. This computation is done by evaluating whether each strand is closed or open: a strand is open if and only if it contains at least one open contact. The algorithm processes each strand from a positive electrical source to a negative electrical source, exploring each possibility when encountering a junction.

When a positive and a negative source can be connected with a closed strand, then the state of each relay on the strand is set to active. Since these new relay states may change the states of the contacts, the strands processing must be reevaluated. So this process is performed until a fixed-point is reached (no relay changed value between two consecutive processing cycles).

Note that, if a fixed-point is not reached, the firmware in the CLEARSY Safety Platform guarantees that the system falls back to a fail-safe state. In the current version of the translator, no check is performed to verify whether this situation might happen. We assume that the design of the relay scheme prevents the oscillation of the electrical circuit and thus ensures the existence of a fixed-point.

Note as well that the transient states of the electrical circuit and of the Clearsy safety platform (before the fixed-point is reached) may be different. The only guarantee is that the fixed-point reached is the same as in the relay circuit.

#### *6.1.3. Verification*

*Verification of the Highlighting.* The highlighting process, being done by hand, is error-prone. We thus developed a tool to compare the result of the highlighting with the original scheme. In addition to the information needed to translate the scheme into B we also save the coordinates of the components. This allows us to re-draw a scheme from the elements highlighted and selected by the users. It is thus easy to compare (by superposition, for example) the two objects to verify that no components or connections have been omitted.

*Verification of the translation.* Two kinds of verification may be performed for the generated B code: a *structural* verification and a *behavioral* verification. A structure based verification is easily achievable since the generated code has a structure that follows precisely that of the intermediate, *ad hoc*, formats used. It would thus be easy, but cumbersome, to compare the two. We envision a tool that would implement a reverse translation from the generated B code back to the strand representation.

Another approach is to verify the behavior of the generated B code against the expected properties of the system. For relay-based schemes, the properties express the expected values of the output after the circuit has stabilized. These properties can then be encoded in the generated B as ASSERT instructions. This results in proof obligations generated in the B environment. Since we are dealing here with finite systems, the proof can be done by exhaustive model checking using ProB. As an example, for a circuit commanding a light, one could require that (even in case of bulb default) the output signal is less permissive than the commanded signal. For instance, the output commanding the green light should not be set if either the orange or the red light are commanded. Conversely, if the orange light is commanded, the output setting the red signal may be set, e.g., if the orange bulb is broken.

Following this approach, we have verified that the code generated for example in Fig. 10 satisfies a number of expected properties provided together with the scheme, expressed in natural language and translated manually to first-order logic.

Model checking was conducted in a few seconds with ProB on this simple industrial design. It is noteworthy that, through such model checking-based verification, we have obtain a counter-example that showed a property was not held (this was expected by the provider of the example). It also helped us identify an early mistake in the translation process (that we have now corrected).

This approach is effective up to midsize designs (as shown by the example in Section 7) but we have not yet applied it to large circuits. However, since the translation targets a subset of the B expression language where types are essentially Booleans, alternative automatic formal verification such as SAT are also applicable and would certainly be able to address larger circuits.

#### *6.1.4. Concluding Remarks*

The work presented shows the feasibility to replace heavy and physically large electrical relay circuits (they need to be fit in a cabinet sideways of the track) by smaller and cheaper digital devices. This approach benefits from the guarantees offered by the application of formal methods (B method, model checking) and from a generic fail-safe device (the CLEARSY Safety Platform).

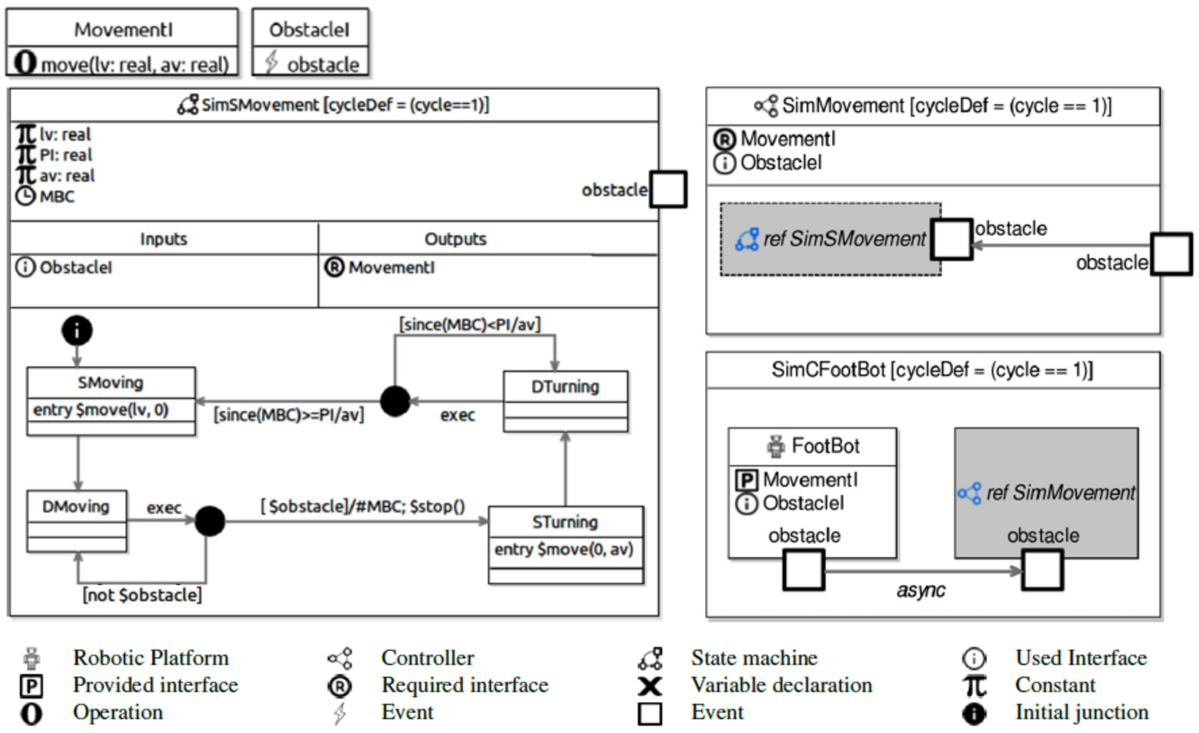


Figure 12: RoboSim: obstacle detection

## 6.2. *RoboSim for Robotics*

RoboSim [6] is a diagrammatic language to model simulations of robotic systems by state machines combined to define concurrent and distributed designs that use specified services of a platform. Its visual representation is akin to notations currently used by practitioners and much more friendly than any programming language. RoboSim main distinction is that its models can be verified against a UML-like design of a controller defined in RoboChart [16]. This is possible because both notations have been given a unified semantics using CSP [17], a process algebra for refinement with well established tools like FDR3 [9]. Hence, by automatically translating their models into CSP and checking for refinement using FDR3, it is possible to automatically check correctness of simulation models regarding their design.

CSP itself has been given a UTP (Unifying Theories of Programming) [10] theory. This allows the encoding of the CSP semantics in the UTP making it possible to obtain support for theorem proving using the powerful prover Isabelle/HOL [8]. In this context, with RoboSim, Cavalcanti et *al.* fully bridge the gap between the state-machine modelling and simulation paradigms. Nevertheless, RoboSim is intended to be an intermediate notation to describe verified simulations that can be automatically translated into code for use with standard robotic simulators. In this section, we present a step towards achieving this using the CLEARSY Safety Platform.

In Fig. 12 we present an illustrative example originally presented in [6] of a RoboSim model of a robot that can move around, detect obstacles, and stop. The module $SimCFootBot$ is composed of the robotic platform $FootBot$ and the $SimMovement$ controller that has a reference to a single simulation machine $SimSMovement$. It is important to notice that the module specifies the cycle period by including a (simple) predicate stating $cycle == 1$. The same happens with the the controller $SimMovement$ and the machine $SimSMovement$.

The interfaces can group variables, operations, and events. In Fig. 12, the interface $MovementI$ has the operations $move(lv, av)$ and $stop()$, provided by the robotic platform, and required by the controller. The operation $move(lv, av)$ can be used to move the robot with linear speed $lv$ and angular speed $av$. The instruction to the robot to stop is given using the operation *stop*. The interface *ObstacleI* has just the event *obstacle*, which is used in the platform, in the controller, and in the state machine. The event *obstacle*, an abstraction of a sensor that detects obstacles, occurs when the robot gets close to any object in its environment. The robotic platform *FootBot* defines the interface of the system with its environment via the operations of the provided interface $MovementI$ and the user interface *ObstacleI*. Assynchronously, the occurrence of the event *obstacle* is sent to the single controller of our example $SimMovement$. The behaviour of a controller is defined by one or more state machines, specifying threads of execution. In our case, the behaviour of $SimMovement$ is defined just by the machine $SimSMovement$. It is important to notice that two different symbols denote an event: the lighting is used when declaring an event, whereas the square is used to indicate event passing information.

State machines are similar to those in UML, except that they have a well-defined action language, and time primitives. The state machine $SimSMovement$ has three local constants $PI$, $lv$, and $av$, and clock $MBC$. The event *obstacle* declared in the interface *ObstacleI* is an input, and the operations *move* and *stop* declared in the interface $MovementI$ are outputs.

A RoboSim model specifies a cyclic mechanism; a special marker event *exec* defines points where behaviour evolution must stop until the next cycle. In each cycle, inputs are read from registers, processed, outputs are written to registers, and then time elapses in a period of quiescence until the next cycle. During processing, the simulation machine takes control of execution until progress requires the (next) occurrence of *exec*.

The visible behaviour is the reading and writing of registers, which is characterised by the inputs and outputs. Their values capture interactions corresponding to platform events, access to platform variables, and calls to platform operations. For instance, the event *obstacle* is captured in our example as a register with a boolean value indicating whether an obstacle has been detected or not. The boolean variable \$*obstacle* corresponding to this input is used in guards, not triggers, of transitions. In RoboSim, the only trigger used is *exec*.

The overall behaviour of $SimSMovement$ is as follows. The first cycle starts with the transition from the initial junction to the $SMoving$ state, in which it is recorded that *move* must be called, as indicated by $move(lv, 0)$. The \$ indicates that the operation is not called immediately. Afterwards, it changes to the $DMoving$ state, where it waits for the next cycle, because there are no transitions from $DMoving$ not triggered by *exec*. In the next cycle, $SimSMovement$ checks whether an obstacle has been perceived. If not, it remains in $DMoving$. Otherwise, it moves to $STurning$, when it resets the $MBC$ clock (denoted by the command $\#MBC$), records that *stop* and then *move* must be called, besides moving to $DTurning$, all in one cycle. In the subsequent cycle, if the amount of time since $MBC$ has been reset is less than $PI/av$, it remains in DTurning; it returns to $SMoving$ otherwise.

### *6.2.1. Translation Overview*

The translation from RoboSim to the CLEARSY Safety Platform, on which we are currently working, must consider the fact that we have two different notions of cycles. On one hand, we have the cycle of the board itself (CLEARSY Safety Platform cycle), which is able to execute around 50 million instructions per second. In each CLEARSY Safety Platform cycle, the board reads the inputs from the input pins and stores their values in reserved input variables, executes the behaviour defined in an special B operation called *user_logic* and writes the values stored in reserved output variables to the output pins. On the other hand, we have the cycle of the simulation model (Model Cycle), which executes one cycle of its state machine possibly reading values from the reserved input variables and writing values to the reserved outputs variables. Conceptually, the time unit of the simulation does not need to be defined. Nevertheless, for execution purposes, we have to provide a definition for that. Our translation defined a constant *cycle_unit*, which must be valuated before loading the project into the board. For the sake of our example, we assigned 100ms to the cycle unit.

A summary of the control flow of B implementation resulting from the translation of RoboSim models is presented in Fig. 13. Initially, the CLEARSY Safety Platform reads all inputs from the pins and stores their values in reserved input variables. Next, we have to check if this is the first time that the *user_logic* is being executed. This is because RoboSim models do not wait one model cycle to provide its first outputs,

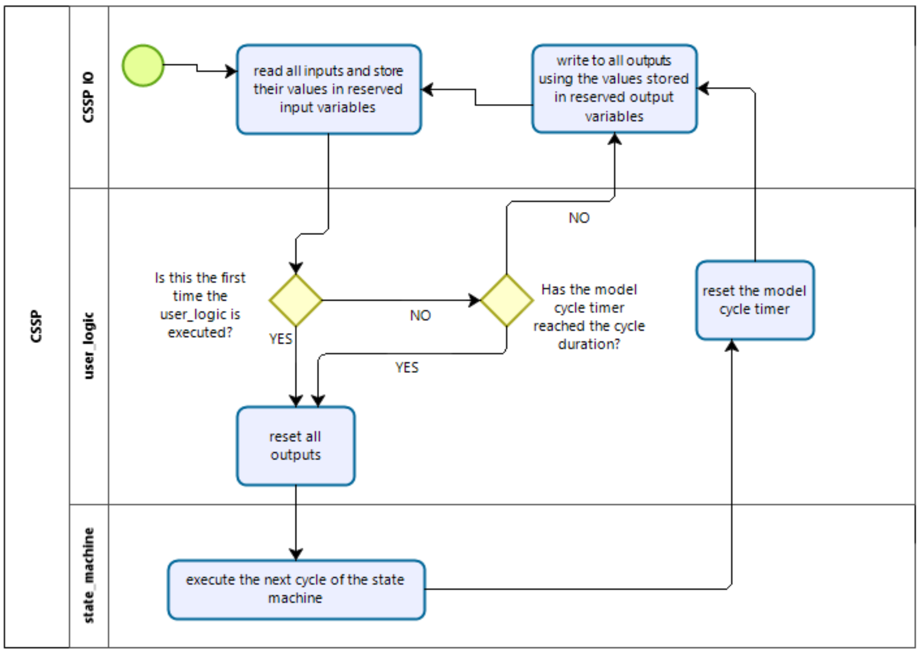
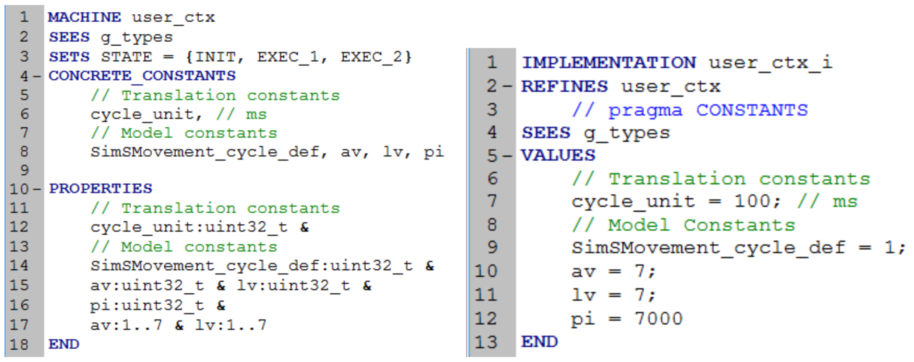

CSSP
CSSP IO
user_logic
state_machine
read all inputs and store their values in reserved input variables
write to all outputs using the values stored in reserved output variables
NO
Is this the first time the user_logic is executed?
NO
Has the model cycle timer reached the cycle duration?
reset the model cycle timer
YES
YES
reset all outputs
execute the next cycle of the state machine

Figure 13: A Summary of the resulting B implementation control flow

which must be given immediately if the simulation model says so. For example, in Fig. 12, the model determines that, initially, the controller must invoke the operation $move(lv, 0)$ before waiting for the next cycle ($exec$). For this reason, in the control flow presented in Fig. 13, if the $user\_logic$ is being executed for the first time, we proceed to the execution of one cycle of the controller state machine. Nevertheless, for reasons we will present later in this section, every such execution must be preceded by a reset of all outputs. Finally, we start a timer that counts the model cycle time and the CLEARSY Safety Platform writes to all output pins using the values stored in reserved output variables that might have had their values changed in the execution of state machine cycle. However, if we are not executing the $user\_logic$ for the first time, we proceed to the execution of one cycle of the state machine only if the model cycle timer has reached the cycle duration. Again, we precede this execution with the outputs being reset and, afterwards, the CLEARSY Safety Platform writes to all output pins using the values stored in reserved output variables. Finally, if the cycle duration has not been reached, the CLEARSY Safety Platform simply writes to all output pins using the values stored in reserved output variables. In fact, the vast majority of the board cycles are empty cycles in the sense that they ignore the inputs being read and do not change any written output.

Another important aspect is that a fine tuning of the model cycle unit is essential to make inputs noticeable by the controller and to make outputs noticeable to the robotic platform. For example, a long model cycle degrades the time between readings of the obstacle sensor and a short model cycle can make it impossible for the car engine to react to the command. Further fine tuning is also necessary in the definition of the values of each of the model constants, namely $lv$, $av$, and $pi$. All constants are specified in a separate context B machine that specifies the properties of these constants. The values of the constants are defined in an implementation B component that refines this context; hence, the B method ensures that the values assigned to all constants satisfy their properties declared in the specification. Fig. 14 presents both components of our example.

In Fig. 15 we present the B implementation of the $user\_logic$. In this B implementation, $first\_time$ is a state variable that is initially $TRUE$. Furthermore, $reset\_outputs$ and $state\_machine$ are operations, which set all output variables to $IO\_OFF$ and executes one cycle of the state machine, respectively. Inputs and outputs are not coded with Boolean as a single memory perturbation is able to change one valid state to another. Hence two values have been defined, IO_OFF and IO_ON, both defined on 8 bits such as



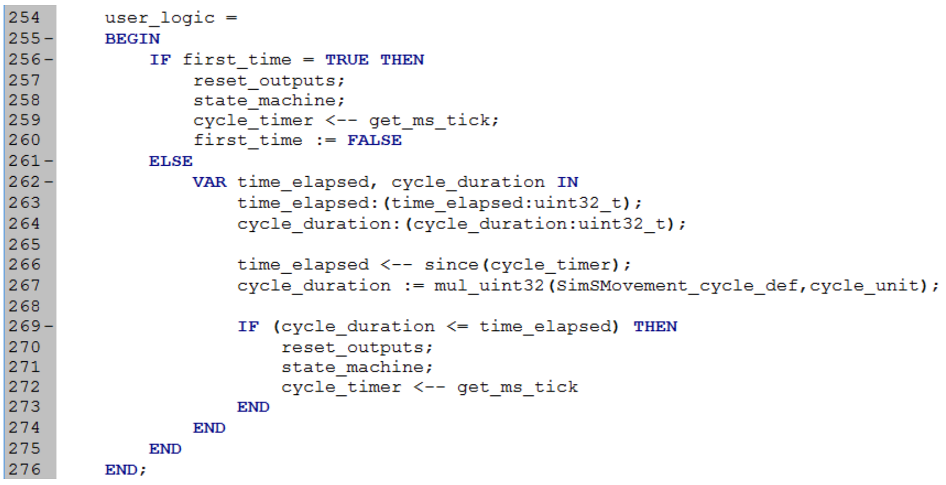
```
MACHINE user_ctx
SEES g_types
SETS STATE = {INIT, EXEC_1, EXEC_2}
CONCRETE_CONSTANTS
    // Translation constants
    cycle_unit, // ms
    // Model constants
    SimSMovement_cycle_def, av, lv, pi

PROPERTIES
    // Translation constants
    cycle_unit:uint32_t &
    // Model constants
    SimSMovement_cycle_def:uint32_t &
    av:uint32_t & lv:uint32_t &
    pi:uint32_t &
    av:1..7 & lv:1..7
END
```

(a) Context Specification

```
IMPLEMENTATION user_ctx_i
REFINES user_ctx
    // pragma CONSTANTS
SEES g_types
VALUES
    // Translation constants
    cycle_unit = 100; // ms
    // Model Constants
    SimSMovement_cycle_def = 1;
    av = 7;
    lv = 7;
    pi = 7000
END
```

(b) Context Implementation

Figure 14: Specification and Implementation of the Simulation Model Constants. The pragma CONSTANTS indicates a context machine impacting the safety and to be checked by the code generation toolchain.



```
254     user_logic =
255     BEGIN
256         IF first_time = TRUE THEN
257             reset_outputs;
258             state_machine;
259             cycle_timer <-- get_ms_tick;
260             first_time := FALSE
261         ELSE
262             VAR time_elapsed, cycle_duration IN
263                 time_elapsed:(time_elapsed:uint32_t);
264                 cycle_duration:(cycle_duration:uint32_t);
265
266                 time_elapsed <-- since(cycle_timer);
267                 cycle_duration := mul_uint32(SimSMovement_cycle_def,cycle_unit);
268
269                 IF (cycle_duration <= time_elapsed) THEN
270                     reset_outputs;
271                     state_machine;
272                     cycle_timer <-- get_ms_tick
273                 END
274             END
275         END
276     END;
```

Figure 15: B Implementation of the *user_logic*

it is very unlikely that a memory corruption leads to the other valid state. If one output is assigned a value that is different from {IO_OFF, IO_ON} then the CLEARSY Safety Platform enters panic mode. In order to reset the cycle timer (lines 259 and 272), stored in the state variable *cycle_timer*, we use the operation *get_ms_tick*, which gives us the current time in miliseconds. To check whether the timer has reached the model cycle duration, we compare the value of the cycle duration (*cycle_duration*) with the time elapsed in the current cycle (*time_elapsed*). The former is the result of multiplying (*mul_uint*32) the constant *SimSMovement_cycle_def*, which is specified in Fig. 14 and corresponds to the cycle duration of the *SimSMovement* state machine defined in Fig. 12, with the cycle unit defined in the fine tuning of the implementation in Fig. 14, which is 100ms. The latter can be obtained using the operation *since*, which receives the value with which the cycle timer has been initialised in the last time it has been reset and returns the difference between this value and the current time, once again using the operation *get_ms_tick*. Finally, the operation *state_machine*, implements the execution of the controller state machine. In our example, we have a single controller state machine. Nevertheless, RoboSim models can have many state machines with different cycle duration each. Our approach naturally deals with this possibility by using different constants for each state machine cycle duration.

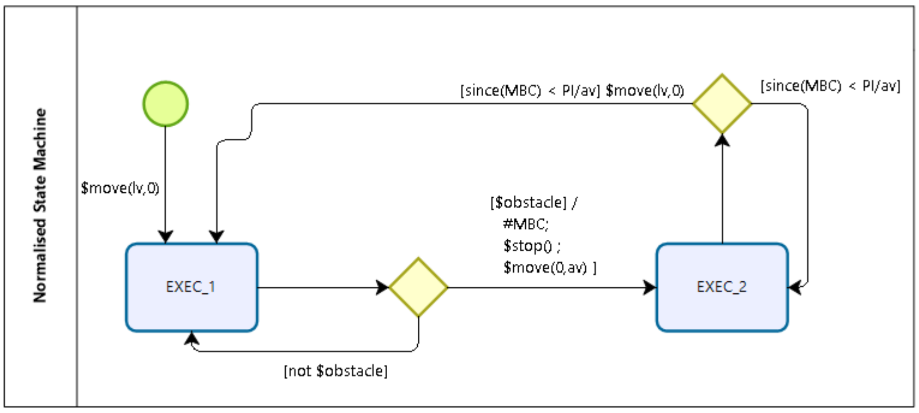


Figure 16: normalized State Machine

*6.2.2. Translating the Controller State Machine*

In general, a translation of state machines into B is not challenging. Nevertheless, unless marked with the special marker event *exec*, RoboSim state transitions are timeless. This important characteristic would not be respected if we simply translate RoboSim models using a straightforward translation because it imposes a wait of at least one model cycle between state transitions.

Our solution is to normalise the states with respect to the model cycles. The state machine resulting from this normalisation has one initial state and one state for each model cycle, which corresponds to the end of transitions marked with the special marker event *exec*. All operation calls of that cycle are composed sequentially and executed in that cycle. For instance, in Fig. 16 we present the result of normalising the state machine of Fig. 12.

In the normalized state machine we only have three states:

- $INIT$: corresponds to the end of the transition leaving the initial junction. In Fig. 16 represented with the circle;
- $EXEC\_1$: corresponds to the end of the transition leaving the state $DMoving$;
- $EXEC\_2$: corresponds to the end of the transition leaving the state $DTurning$.

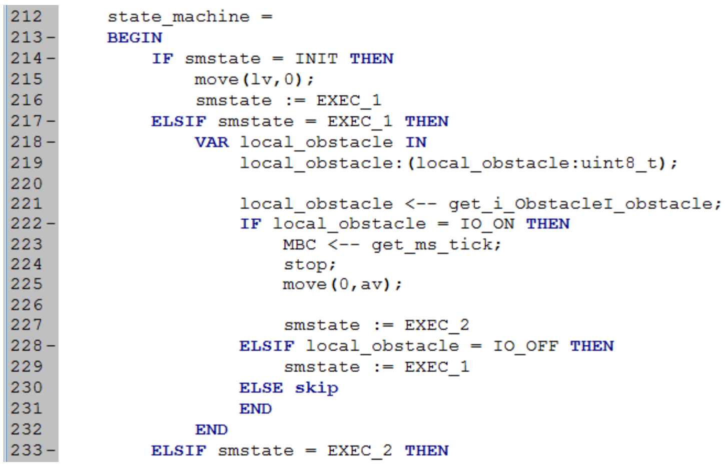
These states are specified as members of an enumerated set, $STATE$, which is declared in the context machine presented in Fig. 14.

Now, the translation of the normalized state machine is relatively trivial. Initially, the code invokes operation $move(lv, 0)$ and enters state $EXEC_1$. In the remaining execution, the resulting code always waits one model cycle before leaving the current state. The main differences of the normalized state machine of our example with that presented in Fig. 12 are:

1. operation calls placed in states, like $\$move(lv, 0)$ originally in state $SMoving$, are now in the state transitions, and
2. the transition from $EXEC_1$ to $EXEC_2$, in which the commands $\#MBC; \$stop()$, originally in the transition from $DMoving$ to $STurning$, and $\$move(0, av)$, originally in the entry of state $STurning$, are sequentially composed in a single transition.

In Fig. 17 we present part of the B implementation of the state machine of our example. All source files can be found at `http://bit.ly/2JtkxuQ`, where you can find the complete Atelier-B project of our running example and an Arduino program that emulates the behaviour of the robotic platform[9]. The *smstate* is a

[9]After creating the CLEARSY Safety Platform project, the only files we have edited are *RoboSim_ctx*, *RoboSim_ctx_i*, *logic* and *logic_i* (the main file, in which all operations mentioned in this paper can be found)



```
212      state_machine =
213-     BEGIN
214-         IF smstate = INIT THEN
215              move(lv,0);
216              smstate := EXEC_1
217-        ELSIF smstate = EXEC_1 THEN
218-             VAR local_obstacle IN
219                  local_obstacle:(local_obstacle:uint8_t);
220
221                  local_obstacle <-- get_i_ObstacleI_obstacle;
222-                 IF local_obstacle = IO_ON THEN
223                      MBC <-- get_ms_tick;
224                      stop;
225                      move(0,av);
226
227                      smstate := EXEC_2
228-                 ELSIF local_obstacle = IO_OFF THEN
229                      smstate := EXEC_1
230                  ELSE skip
231                  END
232              END
233-        ELSIF smstate = EXEC_2 THEN
```

Figure 17: B Implementation of the State Machine

state variable that is initialised with $INIT$. Hence, the first time this operation is invoked, this machine invokes the operation $move(lv, 0)$ and updates the $smstate$ variable to $EXEC_1$. The control returns to the operation *user_logic*, which only invokes the *state_machine* after it reaches the cycle duration. Now, this operation uses the CLEARSY Safety Platform operation *get_i_ObstacleI_obstacle* to get the value of the reserved input variable, *obstacle*. As a standard, we prefix the name of all inputs like *obstacle* with an $i$ and the name of its interface. For example, *i_ObstacleI_obstacle* corresponds to the the input signal *obstacle* of the interface *ObstacleI*. If the value retrieved is $IO\_ON$, the clock $MBC$, implemented as a state variable, is reset (line 223), the operations *stop* and $move(0, av)$ are invoked in this order (lines 224 and 225) and the state machine remains in the current state, $EXEC\_1$.

### 6.2.3. Translating Operation Calls

As previously presented, the operation calls of RoboSim models are directly translated to the invocation of operations of the B implementation. In order to follow the RoboSim semantics presented in [6], we need to consider, for each operation of a RoboSim model, a boolean output value and the operation output values. The former indicates that the operation has been invoked in the current cycle.

An important restriction is the number of input and output pins available in the CLEARSY Safety Platform. In its current version, SK1, the board provides 20 inputs and 8 outputs. During the project creation, we configure the CLEARSY Safety Platform board by mapping each pin to the corresponding input/output. Fig. 18 presents the mapping we have implemented in our example. The first input pin is used to receive the only input, *obstacle*. Our translation uses one output pin for each output operation to indicate that it has been invoked: the output 1 indicates that *move* has been invoked and output 8 indicates that *stop* has been invoked. Similarly to the inputs, our standard prefixes the name of all outputs like *move* with an $o$ and the name of its interface. For example, *o_MovementI_move* corresponds to the invocation of the output operation *move* from the interface *MovementI*. Finally, we are left with six output pins which are used to output the values of $lv$ (pins 2, 3 and 4) and $av$ (pins 5, 6 and 7). For output arguments, we use the name of the argument and the index of the bit as suffixes. For example, $o\_MovementI\_move\_lv\_0$ corresponds to the least significant bit of the argument $lv$ of the operation *move*.

The limitation on the number and type of outputs imposes a property of the constants used in the model. Both, $lv$ and $av$, can only receive natural values ranging from 0 to 7. This platform restriction is included in the $PROPERTIES$ clause of the context machine presented in Fig. 14, in which we include the predicate $av : 0..7 \ \& \ lv : 0..7$. As for all other constants, the B method ensures that the values assigned to these constants in the B implementation satisfy these properties.

By way of illustration, in Fig. 19, we present the implementation of the operations *stop* and *move*. The former implements a parameterless model operation; hence, it simply indicates that the operation

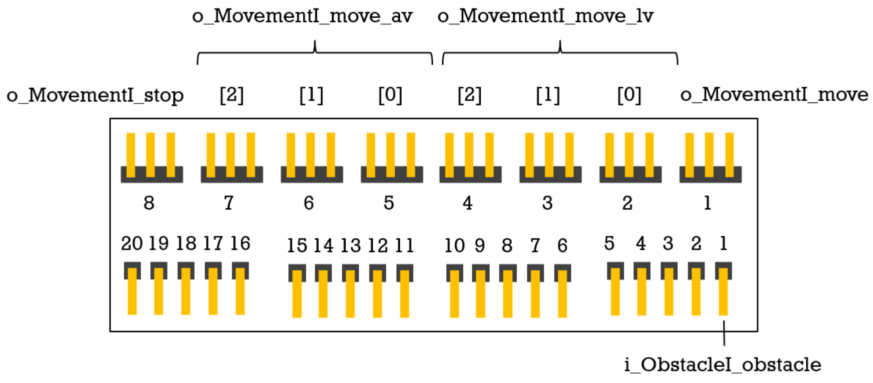
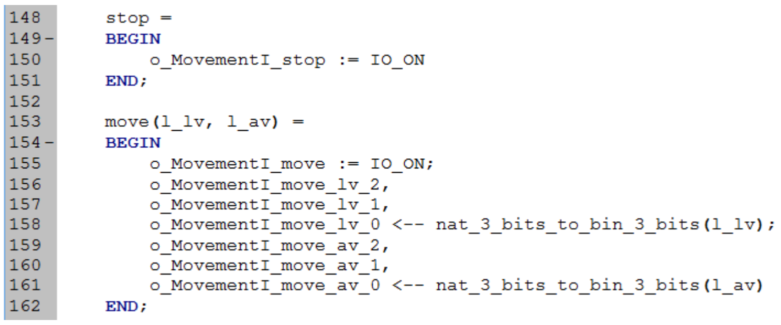


Figure 18: CLEARSY Safety Platform Inputs and Outputs



```
148     stop =
149 -   BEGIN
150         o_MovementI_stop := IO_ON
151     END;
152
153     move(l_lv, l_av) =
154 -   BEGIN
155         o_MovementI_move := IO_ON;
156         o_MovementI_move_lv_2,
157         o_MovementI_move_lv_1,
158         o_MovementI_move_lv_0 <-- nat_3_bits_to_bin_3_bits(l_lv);
159         o_MovementI_move_av_2,
160         o_MovementI_move_av_1,
161         o_MovementI_move_av_0 <-- nat_3_bits_to_bin_3_bits(l_av)
162     END;
```

Figure 19: Operations *move* and *stop*

has been invoked by assingning $IO\_ON$ to the reserved output variable that corresponds to the operation *stop*, $o\_MovementI\_stop$ (line 150). The latter, however, implements a model operation with arguments. For this reason, besides indicating that the operation has been invoked (line 155) it also assigns the values of the arguments $lv$ and $av$ to the corresponding output pins (lines 156 to 161). A local operation $nat\_3\_bits\_to\_bin\_3\_bits$ is used to convert the natural number ranging from 0 to 7 into a binary number and assigns each of its bits to the right corresponding pin.

The translation from RoboSim to the CLEARSY Safety Platform has proved to be an interesting subject and application of CLEARSY Safety Platform for robotic platforms. We are currently working on more elaborated and complex simulation models that will validate our current translation strategy and raise the need for more complex solutions. For instance, some of the models that are in our translation plans have more than 8 outputs ($SK_1$ board). Nevertheless, it is possible to connect different boards in sequence in a way that some of the outputs of one board are inputs to a different board. An investigation on how the model behaviour can spread among different boards is in our near future research agenda. A crucial result that will allow the application of our approach in industry is the automation of our translation strategy. This implementation is also in our research agenda and will define the level of user interaction in the translation process. For example, most of the simulation models like our example model, as expected of simulation models, do not define constant values and cycle unit. This, however, is essential to execute the resulting program in the CLEARSY Safety Platform and needs to be given at some point by the user to the translator.

Finally, as for RoboChart and RoboSim, we intend to provide a CSP semantics to our B implementations. By doing so, it will be possible to automatically check correctness of our B implementations regarding their simulation models by checking for refinement using FDR3. Furthermore, this also allows the encoding of the CSP semantics in the UTP making it possible to obtain support for theorem proving using the powerful prover Isabelle/HOL [8].

## 7. Applications from a System-Level Formal Analysis

This section presents how the design of a CSSP-based product may be conducted in a process that originates in the formal analysis of a system design and proceeds with a model-based design realized through decomposition and refinement.

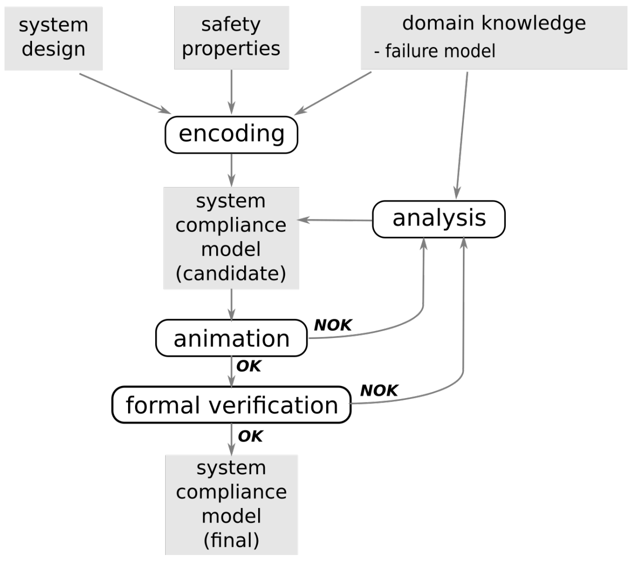


Figure 20: The formal analysis process

The inputs of this process are:

- a system design;
- one or several safety properties that must be ensured by the system;
- domain knowledge;
- identification of the elements of the system that should be implemented in a CSSP-based board;

The formal analysis is based on the first three inputs. Once the formal analysis has been conducted, the last input is used to derive a specification for the CSSP-based component that will be part of the system.

The output of the analysis is a formal model that contains not only the system design logic, but also the required safety properties and all the hypotheses that are necessary to ensure that the system meets the properties. These hypotheses synthesize the domain elements that are necessary to establish the demonstration that the design complies to the safety requirements. These hypotheses must be validated by domain experts. In the case of a fail-safe design, part of the domain knowledge is the possible failure modes of the devices used to implement the system design.

The formal analysis process is pictured in Fig. 20. The process initiates with the construction of a model that encodes the system design, the required safety properties (and possibly the failures). The model is first animated to ensure that it matches the expected behavior. The feedback from this animation might be to include some domain knowledge into the model. A typical domain-oriented constraint would be that a fail-safe sensor does not miss any event that it is supposed to detect. Formal verification is also applied, either through systematic exploration (model checking) or reasoning (proof).

Once the system compliance model has been constructed, the next step is to produce the specification of the system component to be implemented with a CLEARSY Safety Platform board (see Fig. 21). This step is performed using both Event-B [2] and the B method. This step takes as input the system compliance model produced by formal analysis, and the boundaries of the sub-system that shall be implemented as

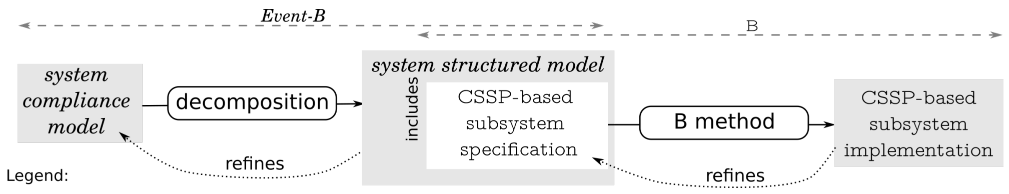


Figure 21: From a System-Level Compliance Model to a CSSP-Based Solution.

a CSSP-based product. These inputs guide the decomposition of the compliance model into a structured model of the system. In this model, all the logic that is to be executed by the CSSP-based subsystem is factored into a B machine. This decomposition is conducted using the Event-B modeling language, extended to allow component structuring constructions.

In the realm of the B method, the machine thus obtained is the *specification* of the function to be executed by the CLEARSY Safety Platform. It is the starting point of the refinement based design approach of the B method, which we use to obtain an implementation capable of being compiled and uploaded to a CLEARSY Safety Platform board (see fig 9).

We have applied this approach for an interlocking system for the railways: a temporary wrong-way interlocking (see Fig. 22). This is a system that is used to manage a track that is temporarily shared between two lines, when a portion of one line is to be temporarily closed. It is composed of two temporary stations, A and C, located at each end of the shared track portion, as well as several sideways equipment, fixed and temporary. The safety property is the absence of front collision on the track. The given system

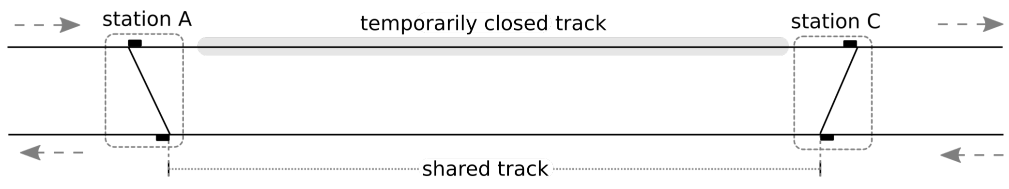


Figure 22: Provisional Wrong-Way Installation

design was a relay-based solution for both stations A and C (presented in [3]). We applied the approach described in this section to build a B module implementing most elements composing station A and derived an implementation compatible for a CSSP-based solution. In practice this would allow to safely replace the expensive, heavy cabinet of interconnected relays, that has to be installed sideways the track nowadays, with a much smaller and lighter fail-save electronic device based on the CLEARSY Safety Platform.

## 8. Conclusion and Perspectives

*Exploitation.* The CLEARSY Safety Platform, combined with improved proof performance and connection with Domain Specific Languages, pave the way to easier development of SIL4 functions (including both hardware and software). The platform safety being out of reach of the software developer, the automation of the redundant binary code generation process and the certificates already obtained for products embedding CLEARSY Safety Platform building blocks, would enable the repetition of similar performances without requiring highly qualified engineers. The CLEARSY Safety Platform building blocks have been used in successive projects where these building blocks have been modified / improved to fulfill diverse requirements. Even if complete cost reduction figures are not yet available, our findings are that software development and certification are reduced by at least 30% as the safety principles do not need to be designed/programmed and as a significant part of the safety case comes from the certification kit (a set of documents explaining how the CLEARSY Safety Platform safety was designed, implemented, tested, and verified, and how the CLEARSY Safety Platform has to be integrated into target hardware - the so-called exported constraints). Moreover, the hardware platform is generic enough to host a large number of complexity-bounded industry applications, with a special focus on the robotics and autonomous vehicles/systems domains. Intelligent road infrastructure also seems of interest, as it appears that fully autonomous cars would require additional support from their environment to deliver a really safe mobility service. This aspect is going to be developed in the coming years.

*Dissemination.* The CSSP IDE is based on Atelier B 4.5.3, providing a simplified process-oriented GUI. It also contains the toolchain to generate the binary, and a bootloader to upload the binary produced on the CSSP board. A first starter kit, $SK_0$, containing the IDE and the execution platform, was released by the end of 2017[10], 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 / guarantee performances, what can or cannot be proved with such systems, etc.
- Less theoretic 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 course when designing exercises, others during these exercises:
  - USB interface is used to program the board and to power it. The second release of the board embeds LEDs to show inputs and outputs status. Many computers do not provide enough current to power all the LEDs, leading to erratic behaviour. The workaround is to power the board with a power supply instead of the USB cable.
  - Time synchronization algorithm between microcontrollers was erroneous. It was not detected during short programming sessions but after leaving boards running during (quite long) coffee breaks.
- 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 inputs and outputs names.
- CLEARSY Safety Platform is yet used to teach in Master 2 in universities and engineering schools. Electronic documentation[11] is used to structure the courses and is updated every 2 months. With three inputs and two outputs, 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).

*Future.* The CLEARSY Safety Platform is a software plant able to generate automatically software for safety critical applications and guarantee its safe execution (outputs are deactivated in case of misbehaviour). This way, it is not required that the developer knows (and masters) all the technical details of the design.

Moreover, the connection of the CLEARSY Safety Platform with domain specific languages, expected to fully hide the formalities, does not perturb the developer in his design activities. The possibility to

[10]https://www.clearsy.com/en/our-tools/clearsy-safety-platform/
[11]Available at https://www.clearsy.com/en/our-tools/clearsy-safety-platform/download-clearsy-safety-platform/

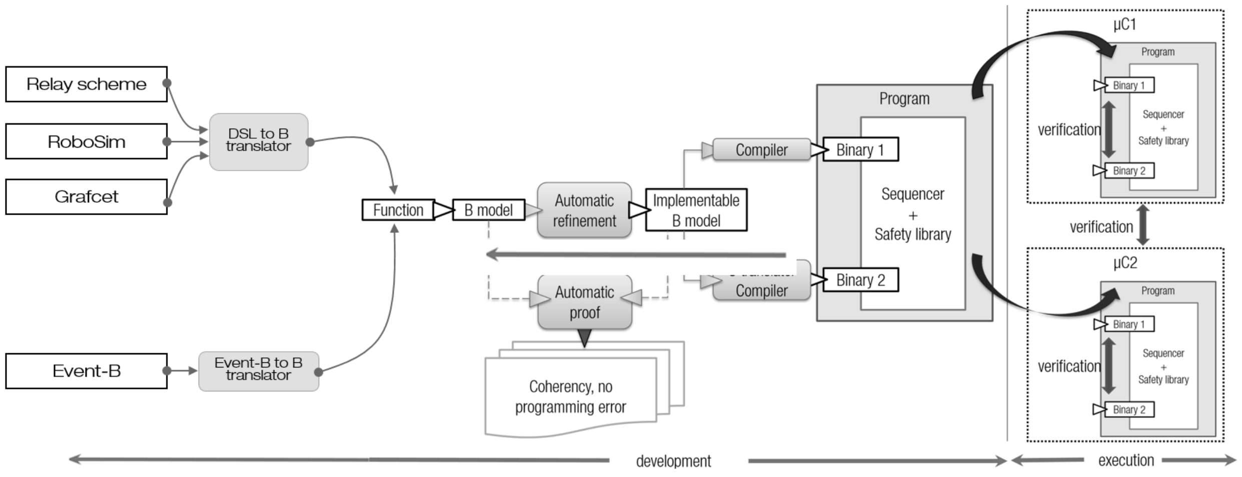


Figure 23: The complete picture including connection with DSLs and system-level proven models. The connection with Grafcet, ongoing, is required to connect with PLCs

derive CLEARSY Safety Platform software specification from a proven system-level specification improves the level of confidence of the final system. Finally the CLEARSY Safety Platform building blocks have been embedded and certified in a number of railway projects in Brazil, Sweden and US, with diverse certification bodies. The CLEARSY Safety Platform is expected to lower the cost of certified safety systems in a number of industrial domains, to contribute to increase citizens safety in our always-more-automated world, and also to convert students and engineers to formal methods due to its ease of implementation.

*Limits of the approach.* The CLEARSY Safety Platform is an innovation combining a number of existing results, many of them issued from previous completed software and electronic projects at CLEARSY. The core of the CLEARSY Safety Platform (software toolchain, core hardware) is certifiable as two notify bodies issued three certificates for railway systems last two years. All the technical justifications are in the 120 pages of the (not public) safety demonstration. The CLEARSY Safety Platform seems competitive up to now as several contracts based on it have been won. However our best experts were involved in its development and first applications. The next systems based on it and developed by "more regular practitioners" will constitute the real test for its acceptance. Similarly the genericity of the platform will be assessed - implemented safety features and design degrees of freedom were designed to adapt to any "plausible" safety system. The extensions in Fig. 23 have not been formally validated. The tools were developed mainly as proofs of concept, to assess if they comply to the 3-U rule: "useful, usable, used". In case of acceptance, stronger scientific work, drafted in this paper, will be required to either validate the existing translation principles or define new ones.

## 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. This research was also partially funded by INES 2.0, FACEPE grant APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0.

[1] Abrial, J.: The B-book - assigning programs to meanings. Cambridge University Press (2005)
[2] Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010), `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569`
[3] de Almeida Pereira, D.I., Déharbe, D., Perin, M., Bon, P.: B-specification of relay-based railway interlocking systems based on the propositional logic of the system state evolution. In: Dutilleul, S.C., Lecomte, T., Romanovsky, A.B. (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11495, pp. 242–258. Springer (2019), `https://doi.org/10.1007/978-3-030-18744-6_16`

[4] Behm, P., Benoit, P., Faivre, A., Meynadier, J.: Météor: A successful application of B in a large project. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I. Lecture Notes in Computer Science, vol. 1708, pp. 369–387. Springer (1999), `https://doi.org/10.1007/3-540-48119-2_22`

[5] Boulanger, J.: Safety of Computer Architectures. Wiley (2010)

[6] Cavalcanti, A.L.C., Sampaio, A.C.A., Miyazawa, A., Ribeiro, P., Filho, M.C., Didier, A., Li, W., Timmis, J.: Verified simulation for robotics. Science of Computer Programming 174, 1–37 (2019), `https://www-users.cs.york.ac.uk/~alcc/publications/papers/CSMRCD19.pdf`

[7] Corbier, F.: Accelerate the development of certified software for train control and monitoring systems. In: in 'Proceedings of 8th European Congress on Embedded Real Time Software and Systems: ERTS 2016 (2016)

[8] Foster, S., Zeyda, F., Woodcock, J.C.P.: Unifying Heterogeneous State-Spaces with Lenses. In: Sampaio, A.C.A., Wang, F. (eds.) Theoretical Aspects of Computing – ICTAC 2016. pp. 295–314. Springer International Publishing, Cham (2016)

[9] Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Model Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)

[10] Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall (1998)

[11] Lecomte, T.: Safe and reliable metro platform screen doors control/command systems. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) FM 2008: Formal Methods, 15th Int'l Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proc. LNCS, vol. 5014, pp. 430–434. Springer (2008)

[12] Lecomte, T.: Applying a formal method in industry: A 15-year trajectory. In: Alpuente, M., Cook, B., Joubert, C. (eds.) Formal Methods for Industrial Critical Systems, 14th Int'l Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proc. LNCS, vol. 5825, pp. 26–34. Springer (2009)

[13] Lecomte, T.: Double cœur et preuve formelle pour automatismes sil4. 8E-Modèles formels/preuves formelles-sûreté du logiciel (2016)

[14] Lecomte, T., Déharbe, D., Prun, É., Mottin, E.: Applying a formal method in industry: A 25-year trajectory. In: da Costa Cavalheiro, S.A., Fiadeiro, J.L. (eds.) Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10623, pp. 70–87. Springer (2017), `https://doi.org/10.1007/978-3-319-70848-5_6`

[15] Lecomte, T., Servat, T., Pouzancre, G.: Formal methods in safety-critical railway systems. In: in 'Proceedings of Brazilian Symposium on Formal Methods: SMBF 2007. pp. 26–30 (2007)

[16] Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J.: Automatic property checking of robotic applications. In: IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 3869–3876 (2017), `https://www-users.cs.york.ac.uk/~alcc/publications/papers/MRLCT17.pdf`

[17] Roscoe, A.: Understanding Concurrent Systems. Springer-Verlag New York, Inc., New York, NY, USA, 1st edn. (2010)

[18] Rétiveau, R.: La signalisation ferroviaire. Presse de l'École Nationale des Ponts et Chaussées (1987), (in French)

[19] Sabatier, D.: Using formal proof and B method at system level for industrial projects. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - 1st Int'l Conf., RSSRail 2016, Paris, France, June 28-30, 2016, Proc. LNCS, vol. 9707, pp. 20–31. Springer (2016)

[20] Wikipedia contributors: Cyclic redundancy check — Wikipedia, the free encyclopedia (2020), `https://en.wikipedia.org/wiki/Cyclic_redundancy_check`, [Online; accessed 08-May-2020]

[21] Wikipedia contributors: Intel hex — Wikipedia, the free encyclopedia (2020), `https://en.wikipedia.org/wiki/Intel_HEX`, [Online; accessed 08-May-2020]

[22] Wikipedia contributors: Ladder logic — Wikipedia, the free encyclopedia (2020), `https://en.wikipedia.org/wiki/Ladder_logic`, [Online; accessed 08-May-2020]

[23] Wikipedia contributors: Programmable logic controller — Wikipedia, the free encyclopedia (2020), `https://en.wikipedia.org/wiki/Programmable_logic_controller`, [Online; accessed 08-May-2020]

[24] Wikipedia contributors: Safety integrity level — Wikipedia, the free encyclopedia (2020), `https://en.wikipedia.org/wiki/Safety_integrity_level`, [Online; accessed 08-May-2020]