# CLEARSY Safety Platform

## Developing Safety Critical Applications

HANDBOOK

CLEARSY Systems Engineering

PUBLISHED BY CLEARSY SYSTEMS ENGINEERING

ISBN: 978-2-9568880-0-0

HTTPS://WWW.CLEARSY.COM/EN/OUR-TOOLS/CLEARSY-SAFETY-PLATFORM/

*Edition February 2020*

# Contents

# 1. Preface

This book provides an introduction to the CLEARSY Safety Platform (CSSP in short). It is aimed at easing the development and the deployment of safety critical applications, up to SIL4. It is made of an integrated software development environment (IDE) and a hardware platform that natively integrates safety principles. It relies on the smart integration of formal methods (including mathematical proof), redundant code generation and compilation, and a hardware platform that ensures a safe execution of the software.

The B formal method is at the core of the software development process. Mathematical proof ensures that the software complies with its specification (functional model) and guarantees the absence of programming errors while avoiding unit testing and integration testing. Moreover only one functional model is used to produce automatically the redundant software, avoiding the need to have two independent teams for its development [1].

The safety principles are built-in, both at software level and at hardware level (2oo2 hardware, 4oo4 software[2]). They are out of reach of the developer who cannot alter them. The detection of any divergent behaviour among the two processors (PIC32 micro-controllers) and the four instances of the software is handled by the platform. The safety verification includes cross checks between software instances and between micro-controllers, memory integrity, ability to control outputs, etc.

R The CSSP Starter kits SK0 and SK1 are only for education and industrial prototyping respectively. If the software generated by the Atelier CSSP is the one that will be running on the final safety critical system, the electronics of SK0 and SK1 do not comply with SIL3 / SIL4 requirements. One of the reasons is that such electronics would largely increase the board prices and it would be clearly against the dissemination of the technology.
However the CSSP Core Module (available by the end of 2019), a daughter board to be plugged on a motherboard with digital IOs, will be SIL4-ready and usable in a real safety critical application.

[1] as required by railways standards for the highest SILs, for examples

[2] put for "2 out of 2" and "4 out of 4", are consensus voting principles used for safety architectures, where all processors have to obtain the results to initiate a potentially dangerous action.

## 1.1 Tool support

Working with the CSSP [3] requires, as a minimum, access to the Atelier CSSP, an IDE derived from Atelier B [4] and extended with specific features like diverse code generation and CSSP starter kit configuration. This IDE allows to create a CSSP project, specify and implement the behaviour of the CSSP, typecheck and compile the project. Finally the IDE allows to upload the program on the CSSP and to monitor its execution. For the execution of the program, either CSSP starter kit 0 ($SK_0$) or 1 ($SK_1$) board may be used. The only difference between the two boards is the number of digital IOs (5 for $SK_0$, 28 for $SK_1$).

## 1.2 Who this book is for

This book is intended primarily as a textbook for post-graduate courses . It is also appropriate for courses on formal methods in general and on (safety related) embedded systems. The book assumes no prior knowledge of formal methods or of reasoning about programs. However it assumes a previous exposure to logic and a basic ability to manipulate simple logic and set theoretic expressions. No prior knowledge of the modelling language, namely the B language, is required as language elements will be introduced when needed. Moreover, as project skeletons are automatically generated, being able to develop a full B project is not required: programming the CSSP requires one component (the user_logic operation) to be modified and possibly new components to be added.

## 1.3 To the instructor

This book has grown out of a course based around the B method and the CLEARSY Safety Platform, developed over a period of three years in Brazil, Canada, France, Italy, Portugal and UK. The material is organised to introduce the central ideas as quickly as possible. This allows the students to become familiar with the tool support at the earliest possible stage, and to use it to develop their own programs. It also means that the students learn the B-method from a software engineering point of view, as they are taught from the viewpoint of using the B-method,instead of the theory beneath. Such theory and language elements are introduced as and when they are needed. The Atelier CSSP generates a pre-filled B project, so the students do not need to develop a full B project but only to complete the specification and implementation of some CONSTANTS, VARIABLES, and OPERATIONS. The first part of the book introduces the overall picture, the development process, and the language elements. They have to be read in this order. The second part of the book contains a number of examples. The first two are related to synchronous and combinatorial programming, they have to be completed before moving to the next ones.

For any further information, requests or questions, please contact:

contact-csp@clearsy.com

## 1.4 Book organisation

This book has an associated website, accessible from

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

This website contains the source code of all the examples and exercises presented in this book.

[3] https://www.clearsy.com/en/our-tools/clearsy-safety-platform/
[4] https://www.atelierb.eu/en/

## 1.5 Acknowledgements

The CLEARSY Safety Platform is being developed to compete on the international scene of safety critical systems, with the key idea to lower development and certification costs. Its development is a collective effort being produced in-house during development but also on site all over the world during deployment and exploitation, to obtain finally a safety, SIL4-ready product.

We would like to thank people involved in the dissemination of the CLEARSY Safety Platform allowing us to deliver talks, courses, and hands-on sessions to their colleagues either teachers, researchers, or students, and in particular: Alexander Romanovsky (Univ. Newcastle, UK), Christiano Braga (UFF, Brazil), Emmanuel Chailloux (LIP6, France), Marc Frappier (Univ. Sherbrooke, Canada), Idir Ait Sadoune (CentraleSupelec, France), José Oliveira (Univ. Minho, Portugal), Leopoldo Teixeira (Univ. Pernambuco, Brazil), Marcel Oliveira (UFRN, Brazil), Tiago Massoni (Univ. Campina Grande, Brazil), Valerio Medeiros (IFRN, Brazil), and Yamine Ait-Ameur (Enseeiht, France).

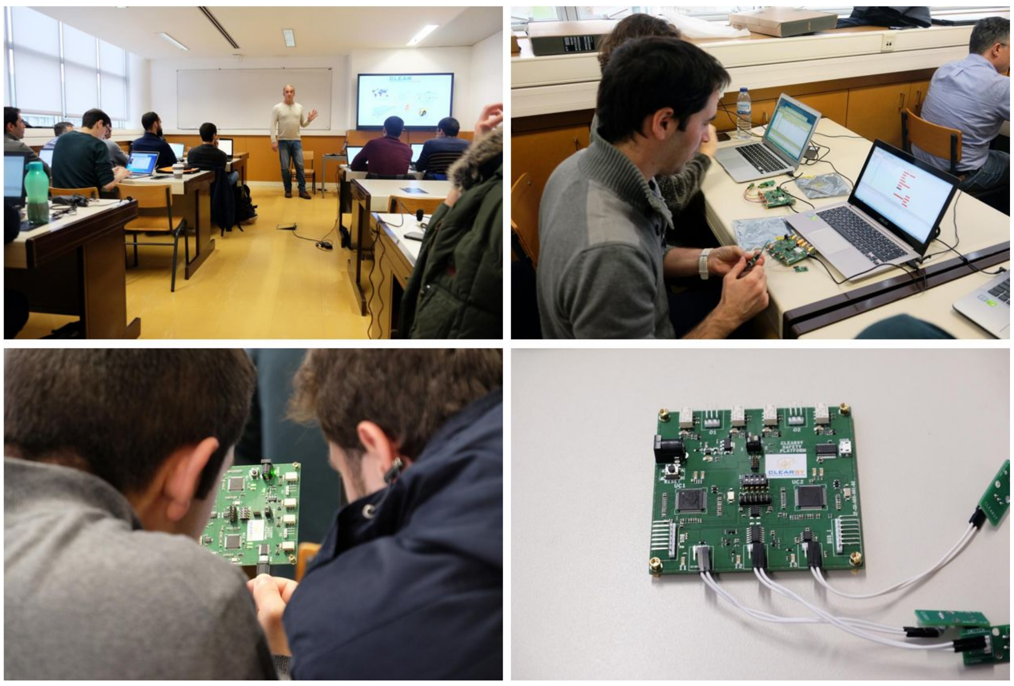


Figure 1.1: A busy hands-on session at the University of Minho in Braga, Portugal

Particular thanks are due to the team in charge of its development over the past years: Adrien Somoza, Bruno Lavaud, David Deharbe, Denis Sabatier, Eliott Trotebas, Emine Aktepe, Etienne Prun, Florent Patin, Guillaume Pressouyre, Hector Ruiz Barradas, José Tarsitano, Lilian Burdy, Loïc Claudet, Ludovic Delfau, Manfred Winkler, Mathieu Comptier, Maxime Renaud, Patrick Sauvage, Sébastien Agostini, Sylvain Breux, Thierry Lecomte, Thomas Gonthier, Vivien Galuchot.

Aix en Provence, France, February 2020

# 2. Introduction

Developing safety critical applications often requires rare human resources to complete successfully, while off-the-shelf block solutions appear difficult to adapt especially during short-term projects. The CLEARSY Safety Platform fulfils a need for a technical solution to overcome the difficulties of developing SIL3/SIL4 system with its technology based on a double-processor and a formal method with proof to ensure safety at the highest level[Lec16]. The formal method, namely the B method[Abr96], has been heavily used in the railway industry for decades[Lec09][Lec08][Ben11]. Using its IDE, Atelier B, to program the CLEARSY Safety Platform ensures a higher level of confidence in the generated software.

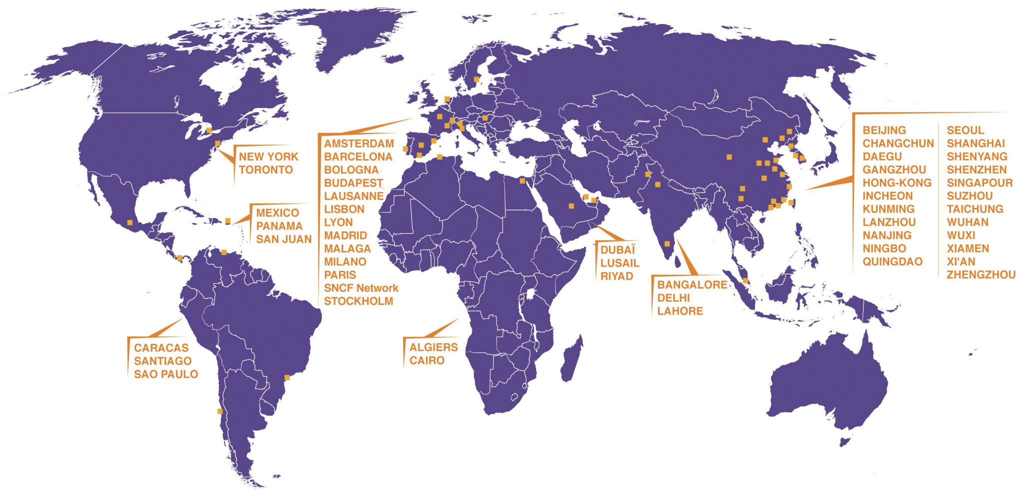


Figure 2.1: Metros and trains equipped with B SIL4 software

The CLEARSY Safety Platform is both a software and a hardware platform aimed at designing and executing safety critical applications. One formal modelling language is used to program the board. Programs are developed using the dedicated IDE or could be the by-product of some translation from a Domain Specific Language to B. The IDE takes care of the verification of the software (type check, proof, compilation) and then ensures its upload to the hardware platform. The program is guaranteed to execute until a misbehaviour is detected, leading to a safe restricted mode where board outputs are deactivated.

The CLEARSY Safety Platform eases the development of safety critical applications as:
- it covers the whole development cycle,
- the safety principles are built-in and are out of reach of the developer, who cannot alter them,
- it is based on a formal language (B) and related proof tools,
- the mathematical proof replaces unit and integration testing.

The CLEARSY Safety Platform eases the certification of safety critical applications as:
- the safety cannot be altered by the developer,
- it will come with a certification kit.

The building blocks of the CLEARSY Safety Platform, already certified in international projects during the years 2017 and 2018 by several certification bodies, have been used to develop a generic version of this technology that could fit a broader range of applications.

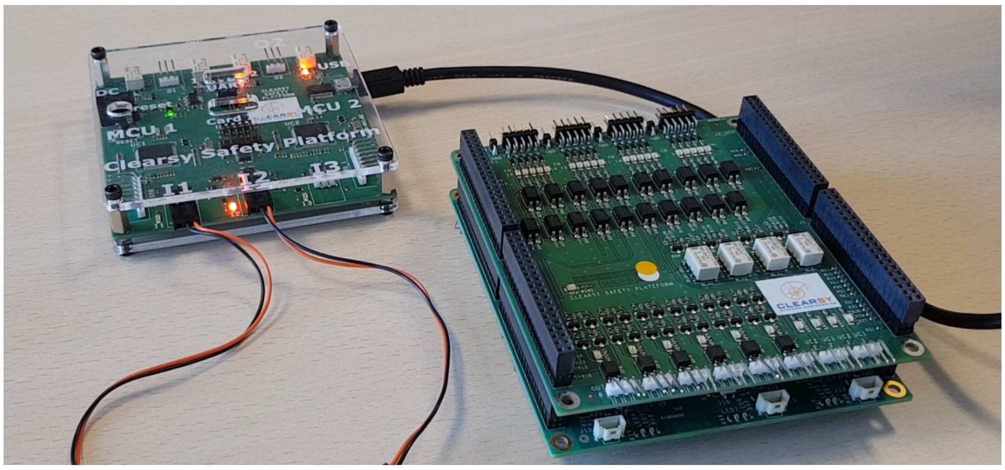


Figure 2.2: Starter kits $SK_0$ (left) and $SK_1$ (right)

The first starter kit, $SK_0$, allows to experiment with the whole development chain, including the IDE, using the B-Method and an electronic board hosting the safe execution platform relying on two PIC32 microcontrollers, providing 3 digital inputs and 2 digital outputs.

The second starter kit, $SK_1$, is functionally identical to $SK_0$. It provides 20 digital inputs and 8 digital outputs. The core automaton, with its two PIC32 microcontrollers, is hosted on a motherboard while the inputs/outputs are located on a daughterboard.

# Description

# 3. Architecture and Safety Principles

## 3.1 Introduction to safety

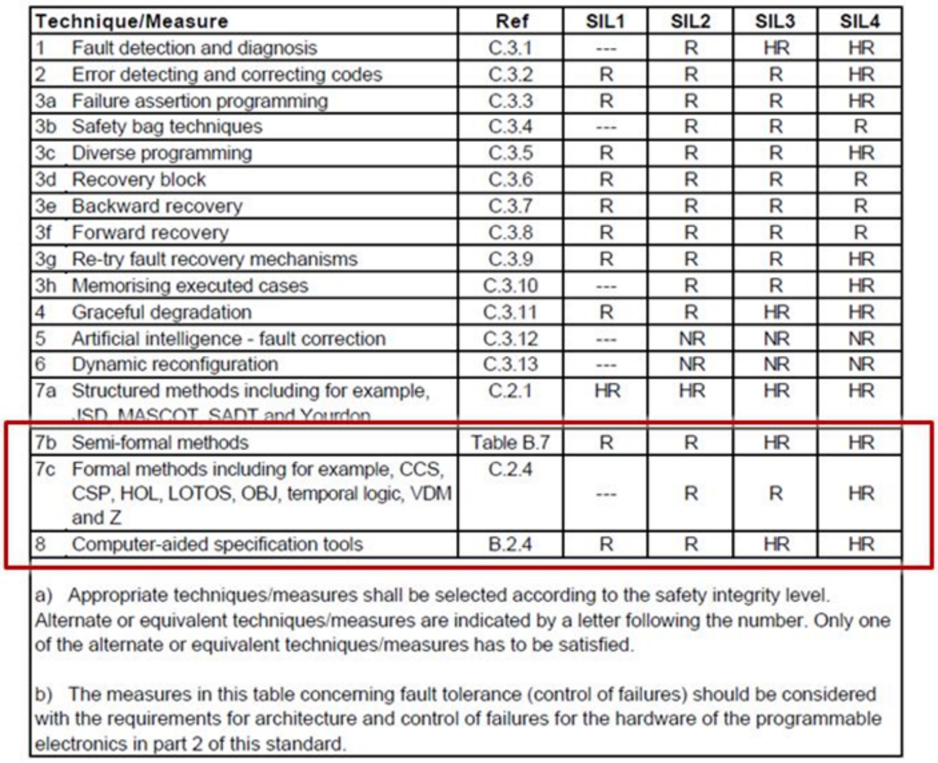
Safety critical systems are systems where life is at risk. One *mistake* could lead to injury or death (of passengers aboard trains, planes, or cars for example).

One question for a developer is:

"would you dare to execute your program if someone would die in case of a crash / core dump / fatal error / etc. especially someone you know like a friend or a member of your family?"

Figure 3.1: Double collision which occurred on 8th October 1952 at Harrow and Wealdstone Station

When you develop a safety system, you are not left alone. Depending on your application domain, you have standards that provide you guidance based on the safety level you are looking for. Note that these standards do not provide a definitive recipe to produce safety systems, but more a collection of state-of-the-art recommendations related to the software, hardware, and development process [1].



| Technique/Measure | Ref | SIL1 | SIL2 | SIL3 | SIL4 |
|---|---|---|---|---|---|
| 1 Fault detection and diagnosis | C.3.1 | --- | R | HR | HR |
| 2 Error detecting and correcting codes | C.3.2 | R | R | R | HR |
| 3a Failure assertion programming | C.3.3 | R | R | R | HR |
| 3b Safety bag techniques | C.3.4 | --- | R | R | R |
| 3c Diverse programming | C.3.5 | R | R | R | HR |
| 3d Recovery block | C.3.6 | R | R | R | R |
| 3e Backward recovery | C.3.7 | R | R | R | R |
| 3f Forward recovery | C.3.8 | R | R | R | R |
| 3g Re-try fault recovery mechanisms | C.3.9 | R | R | R | HR |
| 3h Memorising executed cases | C.3.10 | --- | R | R | HR |
| 4 Graceful degradation | C.3.11 | R | R | HR | HR |
| 5 Artificial intelligence - fault correction | C.3.12 | --- | NR | NR | NR |
| 6 Dynamic reconfiguration | C.3.13 | --- | NR | NR | NR |
| 7a Structured methods including for example, JSD, MASCOT, SADT and Yourdon | C.2.1 | HR | HR | HR | HR |
| 7b Semi-formal methods | Table B.7 | R | R | HR | HR |
| 7c Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z | C.2.4 | --- | R | R | HR |
| 8 Computer-aided specification tools | B.2.4 | R | R | HR | HR |

a) Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

b) The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in part 2 of this standard.

Figure 3.2: Table from IEC 61508 standard showing design recommendations for SIL1 to SIL4 software (R: recommended, HR: highly recommended, NR: not recommended. Note that nothing here, including formal methods, is mandatory

Before connecting your system to the real world and switching it on, you need to complete a safety case, that is a demonstration the feared event(s)[2] will not happen more frequently than expected. For SIL3, it is one failure every 100 years, and for SIL4, one failure every 10,000 years.

For the safety case, you need to consider the whole picture: the hardware (computers, sensors, actuators, etc.), the software, and the environment at large. The main question is:

"what could happen in case that something fails?"

We are far from the concept: "it compiles, hence it works". The safety case always depends on a number of hypotheses that restrict the scope. The idea is not to protect against everything but only to consider a set of "reasonable" situations. As an example, for train collision, we do not consider a plane that could crash on the tracks.

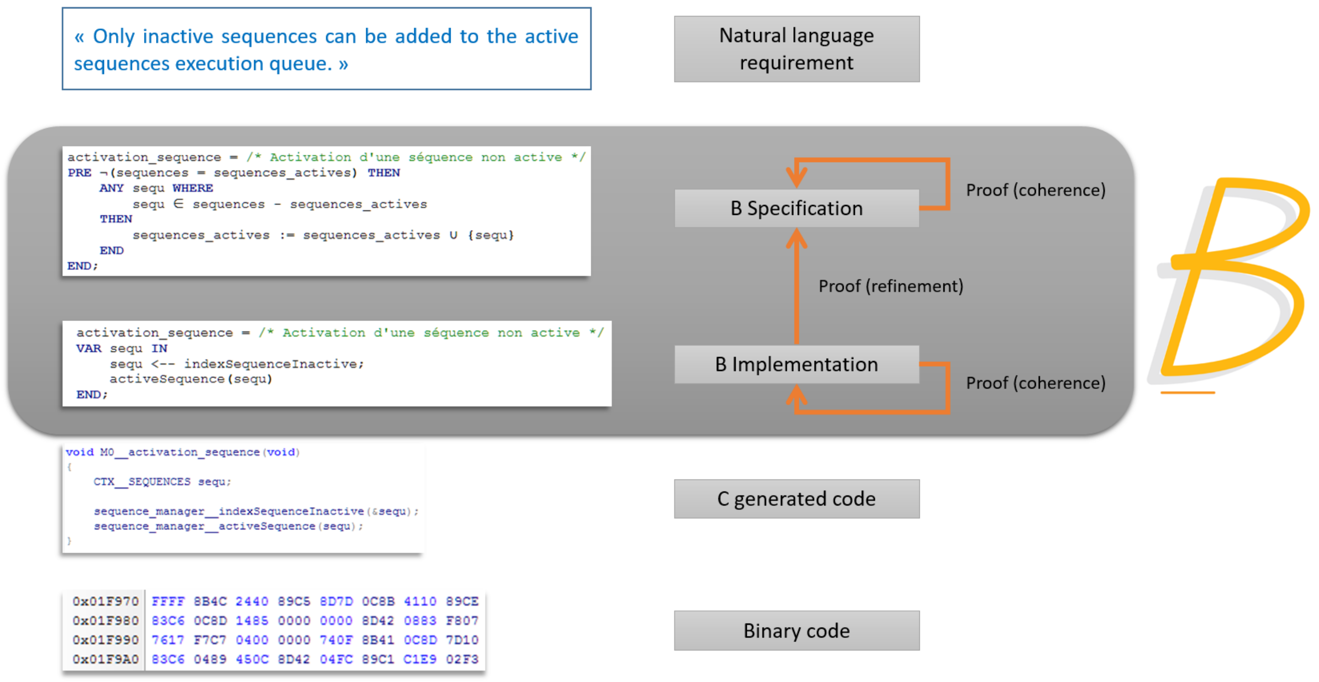
The "failures" considered are diverse and represent a large spectrum of situations:

- specification error: we specified the wrong system or software.
- development error: the specification is correct but the implementation does not comply with its specification. It could be a functional mistake: the algorithm is doing something different from what is expected. It could be non-functional: the operation is performed slower than expected.

[1] and also maintenance
[2] in the railway sector, it is mainly train collision

- programming error: numbers are divided by 0, arithmetic computation produces overflow, or tables are accessed outside of their range.
- compilation error: binary code produced does not comply with source code [3]
- wrong execution: the memory is corrupted (wrong data, wrong instruction, corrupted program counter), the hardware is failing (wrong instruction execution, incorrect storage/access)
- failing hardware: sensors are providing incorrect values, actuators cannot be commanded properly.

For a SIL4 system, the target reliability is between $10^{-7}$/h to $10^{-9}$/h. Given that a processor reliability is estimated between $10^{-4}$/h to $10^{-6}$/h, a single processor is not sufficient for a SIL3 or SIL4 system. That is why two processors (or more) are used in parallel (traditionally with a voter - the two processors have to come to the same decision to initiate a potentially risky action [4]). In addition, the two processors are equipped with protecting mechanisms that would allow the system to continue its mission in case of perturbation / failure.



« Only inactive sequences can be added to the active sequences execution queue. »

```
activation_sequence = /* Activation d'une séquence non active */
PRE ¬(sequences = sequences_actives) THEN
    ANY sequ WHERE
        sequ ∈ sequences - sequences_actives
    THEN
        sequences_actives := sequences_actives ∪ {sequ}
    END
END;
```

```
activation_sequence = /* Activation d'une séquence non active */
VAR sequ IN
    sequ <-- indexSequenceInactive;
    activeSequence(sequ)
END;
```

```
void M0__activation_sequence(void)
{
    CTX__SEQUENCES sequ;

    sequence_manager__indexSequenceInactive(&sequ);
    sequence_manager__activeSequence(sequ);
}
```

```
0x01F970 | FFFF 8B4C 2440 89C5 8D7D 0C8B 4110 89CE
0x01F980 | 83C6 0C8D 1485 0000 0000 8D42 0883 F807
0x01F990 | 7617 F7C7 0400 0000 740F 8B41 0C8D 7D10
0x01F9A0 | 83C6 0489 450C 8D42 04FC 89C1 C1E9 02F3
```

Figure 3.3: Path from requirements to binary code with B

The contribution of B to the safety case is explained in figure 3.3. In this figure, we see the different stages from requirements to binary code (from top to bottom). The B method covers the software specification and implementation stages (the grey box) where the specification and implementation models are proved (we will see later on what it means). However the inputs and outputs of this grey box are error prone:

- the specification model could be different from the natural language requirements. Usually human based cross verification is required - every requirement should be in the model, every modelling element should be issued from the natural language requirements. Validation testing also helps to find mistakes at this stage.
- the code generated could be different from the implementation model: the code generator may alter the semantics of the implementation model due to incorrect code generator specification

[3]Who is reading the binary code these days?

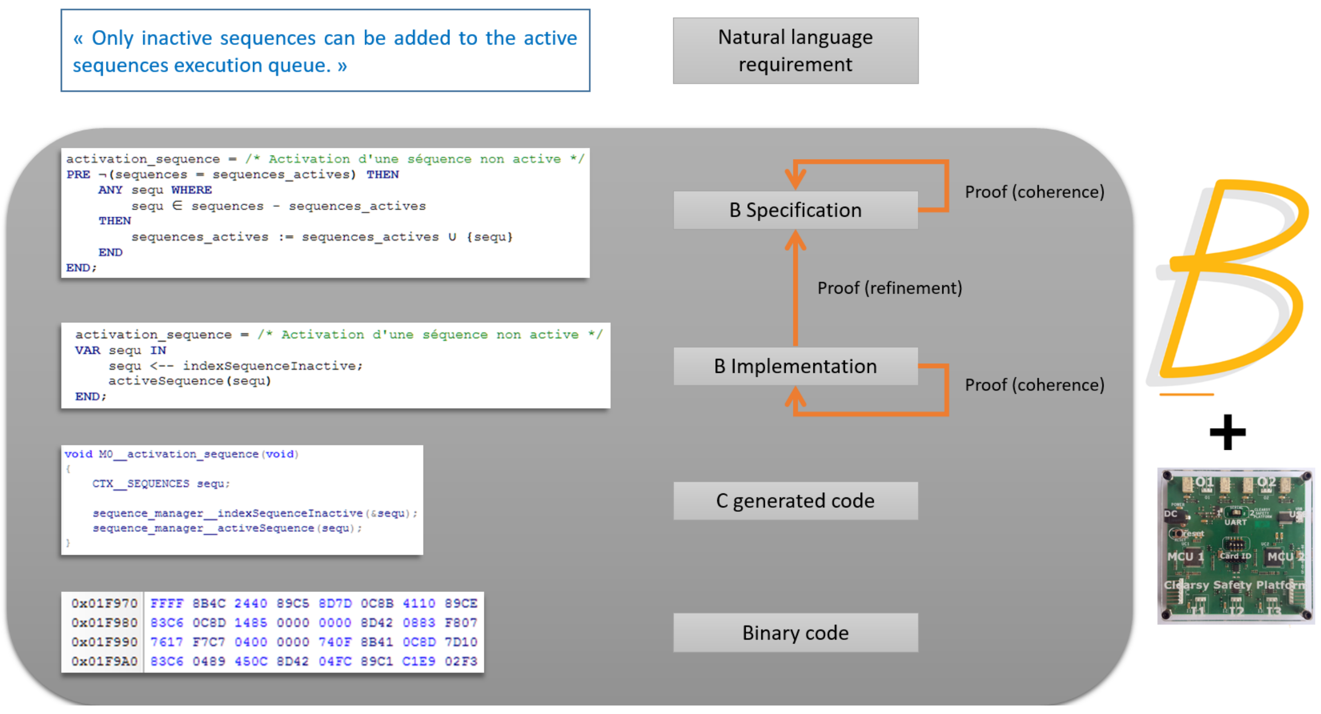
[4]for example, opening platform screen doors on a metro platform with the risk that waiting passengers could fall on the tracks if no train is present.

or bugs.

- the binary code could be different form the source code: the compiler may alter the semantics of the source code, due to improper optimisation options or bugs.
- the processor executing the binary code could exhibit a misbehaviour due to either internal reasons (design or production flaw) and/or external reasons (high energy particles, electro-magnetic waves, etc.).

Usually the last three bullets are taken into account with diversity, given that the program is executed on two (or more) processors: two different code generators are used to produce two different source codes from the same formal model. It is very unlikely that two tools developed with different technologies (programming language, libraries, compiler) and by independent teams are going to exhibit the same unsafe behaviour under the same conditions. One code generator could for example use a big endian memory model while the other uses a little endian one. Another option is to add void instructions [5] in one source code only that do not change the final behaviour but produce a different binary code. That way a processor perturbation would not affect the two programs the same way because different parts of each program are being executed.

We can clearly see that in the case of safety systems, the B method is only one part of the story and other means have to be set up to reach safety related objectives. These means require rare human resources to complete successfully because of the deep level of understanding of both hardware and software parts to consider.



```
activation_sequence = /* Activation d'une séquence non active */
PRE ¬(sequences = sequences_actives) THEN
    ANY sequ WHERE
        sequ ∈ sequences - sequences_actives
    THEN
        sequences_actives := sequences_actives ∪ {sequ}
    END
END;
```

```
activation_sequence = /* Activation d'une séquence non active */
VAR sequ IN
    sequ <-- indexSequenceInactive;
    activeSequence(sequ)
END;
```

```
void M0__activation_sequence(void)
{
    CTX__SEQUENCES sequ;

    sequence_manager__indexSequenceInactive(&sequ);
    sequence_manager__activeSequence(sequ);
}
```

```
0x01F970 FFFF 8B4C 2440 89C5 8D7D 0C8B 4110 89CE
0x01F980 83C6 0C8D 1485 0000 0000 8D42 0883 F807
0x01F990 7617 F7C7 0400 0000 740F 8B41 0C8D 7D10
0x01F9A0 83C6 0489 450C 8D42 04FC 89C1 C1E9 02F3
```

Figure 3.4: Path from requirements to binary code with B with the CLEARSY Safety Platform

With the CLEARSY Safety Platform, the very technical aspects related to safety are taken into account by the platform (see chapter 3.3), leaving the developer to focus only on the development of the function to perform. In the figure 3.4, the combination of B and CSSP covers all the steps

[5] One code could be xx := yy + zz while the other is xx := yy + zz + 1 - 1

from software specification to binary code. The developer is only required to be able to specify and program in B (with DSL if a translator from DSL to B is available), thus less expert profiles could be used for the development. The only remaining activity to perform (apart from validation testing, always mandatory) is the traceability/coverage between natural language requirement and formal modelling. Hence the CLEARSY Safety Platform, with its technology based on a double processor and a formal method with proof to ensure safety at the highest level, fulfils the need for a technical solution to overcome the difficulties of developing SIL3/SIL4 systems.

## 3.2 Architecture

The CLEARSY Safety Platform is made of two parts: an IDE to develop the software and an electronic board to execute this software. The full process is described in figure 3.5.

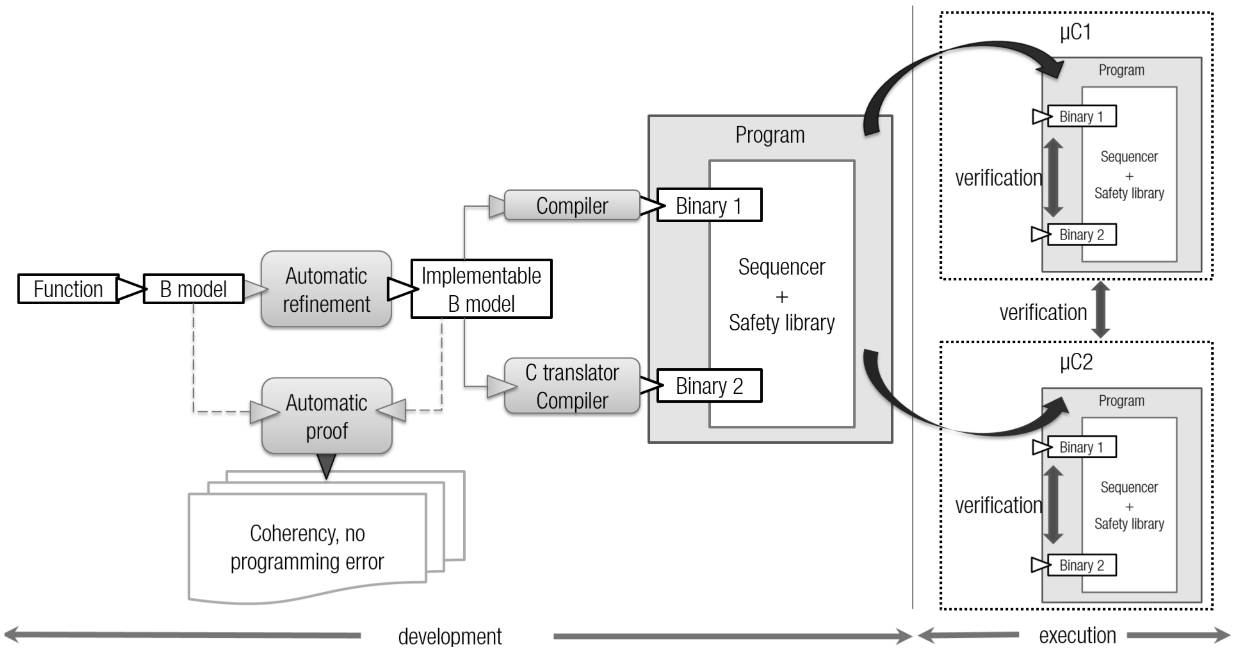


Figure 3.5: Full path from function description to safe execution with the CLEARSY Safety Platform. Round boxes are tools, rectangular boxes are files.

It starts with the function specification (natural language) to develop. The developer has to provide a B model of it (specification and implementation) using the schema:

- the function to program is a loop, where the following steps are performed repeatedly in sequence:
  - the inputs are read [6]
  - some computation is performed
  - 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 implementation is usually handwritten but could also be generated automatically with the B Automatic Refinement Tool [7]. The B models are proved (mostly automatically as the level of

[6]Inputs are similar for $\mu C_1$ and $\mu C_2$, unless the inputs are captured at different times in which case the different values would cause the platform to enter panic mode.

[7]However automatic refinement requires a higher level of experience of B and is not covered in this book.

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:

- $\text{binary}_1$, obtained via a dedicated compiler (developed by CLEARSY) transforming a B model into HEX [8] file,
- $\text{binary}_2$, 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 toolchains. Then the two binaries $\text{binary}_1$ and $\text{binary}_2$ are linked with:

- a sequencer, in charge of reading inputs, executing $\text{binary}_1$ then $\text{binary}_2$, and setting the outputs
- a safety library, in charge of performing safety verification (more details in chapter 3.3). In case of failing verification, the board enters panic mode, meaning the outputs are deactivated [9], the board status LED start 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 $\text{binary}_1$, $\text{binary}_2$, the sequencer and the safety library. The memory mappings of $\text{binary}_1$ and $\text{binary}_2$ are separate.

This program is then uploaded on the two microcontrollers $\mu C_1$ and $\mu C_2$. 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 its execution, the following are performed:

- internal verification:
  - every cycle, $\text{binary}_1$ and $\text{binary}_2$ memory spaces (variables) are compared
  - regularly, $\text{binary}_1$ and $\text{binary}_2$ memory spaces (program) are compared [10]
  - regularly, the identity between memory output states and physical output states is checked to detect if the board is unable to command the outputs.
- external verification:
  - regularly (every 50ms at the latest), memory spaces (variables) are compared between $\mu C_1$ and $\mu C_2$.

If any of these verifications fail, the board enters the panic mode.

The whole process is fully supported by adequate tools. In the figure 3.6, the tools and text/binary files generated are made explicit for both the application (path used every time an application is developed) and the safety belt (developed once for all by the IDE development team [11]. The tools are issued 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 [12].
- the C to HEX GCC compiler.
- the linker combining the 2 hex files with the safety sequencer and libraries.
- the bootloader

[8] "Intel HEX is a file format that conveys binary information in ASCII text form. It is commonly used for programming microcontrollers, EPROMs, and other types of programmable logic devices." (Wikipedia)

[9] No power is provided to the Normally Open (NO) outputs, so the output electric circuits are open.

[10] This verification is performed "in the background" over thousands/millions of cycles, to keep a reasonable cycle time.

[11] Note that from the abstract formal model, one part of the software is developed in B with 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.

[12] In order to ease debugging as ASM MIPS to HEX is a straightforward line-to-line translation.

We can clearly see that the CLEARSY Safety Platform is a generic PLC [13] able to perform command and control over inputs and outputs. The overall architecture is similar among all instances of the CLEARSY Safety Platform. The differences are due to the physical interface:

- 5 IOs for $SK_0$, 28 for $SK_1$.
- digital (Boolean) IOs for $SK_0$ and $SK_1$, analog IOs in the future.
- network connection (messaging) through a maintenance processor, in the future.

R 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 [14] would "only" require the existing electronic board and software tools to be modified, without impacting much the safety demonstration.

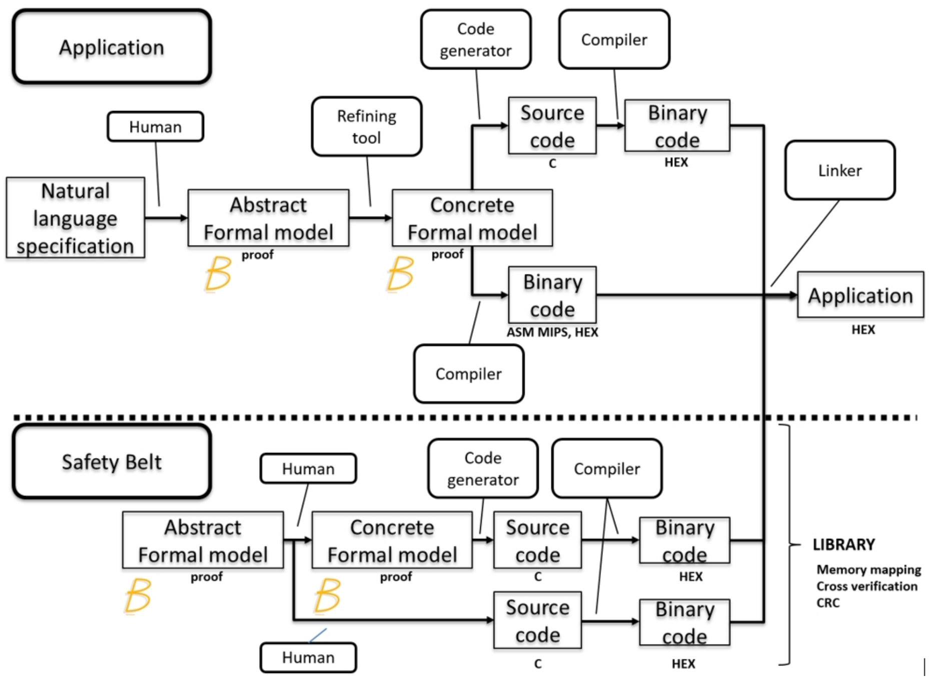


Figure 3.6: Tools and files involved in the generation of the software

## 3.3 Safety Principles

The safety is built on top of few 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,
- four instances of the same function running on two microcontrollers (two per microcontroller 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 the two $\mu C$,

[13] "A Programmable Logic Controller is an industrial digital computer which has been ruggedized and adapted for the control of manufacturing processes, such as assembly lines, or robotic devices, or any activity that requires high reliability control and ease of programming and process fault diagnosis." (wikipedia)

[14] STM32 for example

- outputs require both $\mu C_1$ and $\mu C_2$ to be alive and running as one provides energy and the other one the command,
- output physical states are regularly verified to comply with the memory 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) to prevent the short-circuit current from being considered as high level (permissive) logic.

| **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 | | Implemented behaviour incompatible with invariant properties | Unprovable proof obligation indicates implementation mistake |
| implementation | 4 | Implemented behaviour incompatible with specified behaviour | Unprovable proof obligation indicate implementation mistake |
| implementation | 5 | Overflow capable arithmetic operators used instead of dedicated ones | Detected by the B-to-HEX compiler |
| implementation | 6 | IF clause with more than one condition (B0 language restriction) | Detected by the B-to-HEX compiler |
| implementation | 7 | LOCAL variables not typed before use (B0 language restriction) | Detected by the B-to-HEX compiler |
| code generation | 8 | Syntax errors in the C generated code | Detected by the MICROCHIP compiler |
| code generation | 9 | Incorrect naming in the C generated code | Detected by the linker |
| code generation | 10 | Incorrect memory map | Memory overlap detected by the bootloader |

Table 3.1: Verification performed during development

However, as explained in the chapter 1, the electronic board lacks some vital elements to comply with highest SIL requirements like:

- ensure galvanic isolation between the two half-boards, to prevent that one side of the board wrongly provides energy to the other side's outputs,
- activate safety outputs with a sinusoidal signal [15] instead of a continuous signal, to ignore fault current and activate output.

These missing features are only needed for real-life safety critical systems and do not prevent developers, whether students, researchers or engineers, from using the CLEARSY Safety Platform for education and prototype development.

The verification performed by the CLEARSY Safety Platform, either during development or execution stages, is summarised in tables 3.1 and 3.2.

[15] The microcontroller needs to be alive to generate the sinusoidal signal.

| **Stage** | **#** | **Failure** | **CSSP verification** |
|---|---|---|---|
| compilation | 11 | Wrong binary code generated | Detected during execution by safety library by comparing binary$_1$ and binary$_2$ variables in memory with CRC on the same $\mu C$ |
| uploading | 12 | Incorrect transfer between host and electronic board | Detected by bootloader during upload (CRC) and during execution over several cycles |
| execution | 13 | RAM error (variables) | Detected by comparing binary$_1$ and binary$_2$ variables in memory with CRC on the same $\mu C$ |
| execution | 14 | RAM error (program) | Detected by comparing binary$_1$ and binary$_2$ program in memory with CRC with the other $\mu C$ |
| execution | 15 | Failure of one $\mu C$ | Detected by handshake between $\mu C_1$ and $\mu C_2$ at least every 50 ms |
| execution | 16 | Outputs not command-able | Detected by checking physical state and command issued by the software |

Table 3.2: Verification performed during execution

# 4. Programming

## 4.1 Installation

The starter kit $SK_0$ is composed of:

- The CLEARSY Safety Platform board
- A micro-USB connector to upload the software and to monitor its execution
- A power supply
- 3 switches connected to the 3 digital inputs $I_1$, $I_2$ and $I_3$
- The development environment running on Windows
- The user manual (this book)

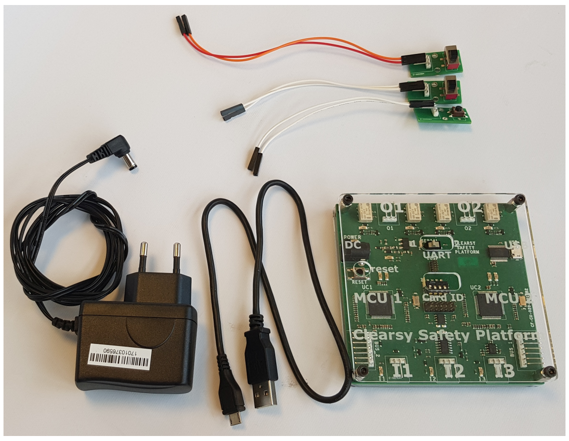
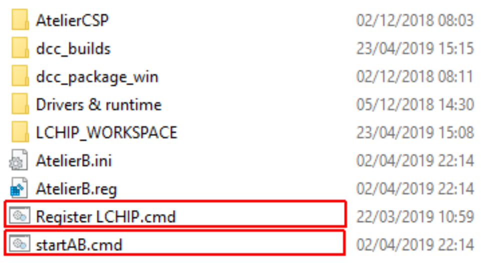


Figure 4.1: The CLEARSY Safety Platform starter kit 0

Warning ! Do not connect the board to your PC before having completed the installation of the FTDI driver. Sometimes Windows installs another (inadequate) driver that prevents any further communication with the board. Then removing this driver is mandatory before installing the FTDI driver.

The installation of the development environment is as follow:

- download the installer file containing the IDE [1]. The file is around 0.4 GB.
- execute the installer and install it to a directory. It requires 3 GB of free space on your hard disk. Choose preferably a path which does not contain spaces or special characters.
- install the FTDI driver. Execute the file CDM21228_Setup.exe in the directory "Drivers & runtime". This driver is required to emulate the CSSP serial port over USB in order to communicate with the board (upload, monitor).



| | |
|---|---|
| AtelierCSP | 02/12/2018 08:03 |
| dcc_builds | 23/04/2019 15:15 |
| dcc_package_win | 02/12/2018 08:11 |
| Drivers & runtime | 05/12/2018 14:30 |
| LCHIP_WORKSPACE | 23/04/2019 15:08 |
| AtelierB.ini | 02/04/2019 22:14 |
| AtelierB.reg | 02/04/2019 22:14 |
| Register LCHIP.cmd | 22/03/2019 10:59 |
| startAB.cmd | 02/04/2019 22:14 |

Figure 4.2: The directory where the IDE has been installed. The two scripts to use for respectively configuring and starting the IDE are highlighted in red. Register LCHIP.cmd is executed automatically during installation. startAB.cmd launches the CSSP IDE.

## 4.2 A first run

During this first run, we are going to experiment with the full process of programming the CLEARSY Safety Platform. Let us start by executing the startAB.cmd script to verify that the installation has been completed. You should obtain the window shown in the figure 4.3 with the two projects *Clock* and *Combinatorial* listed in the left pane.

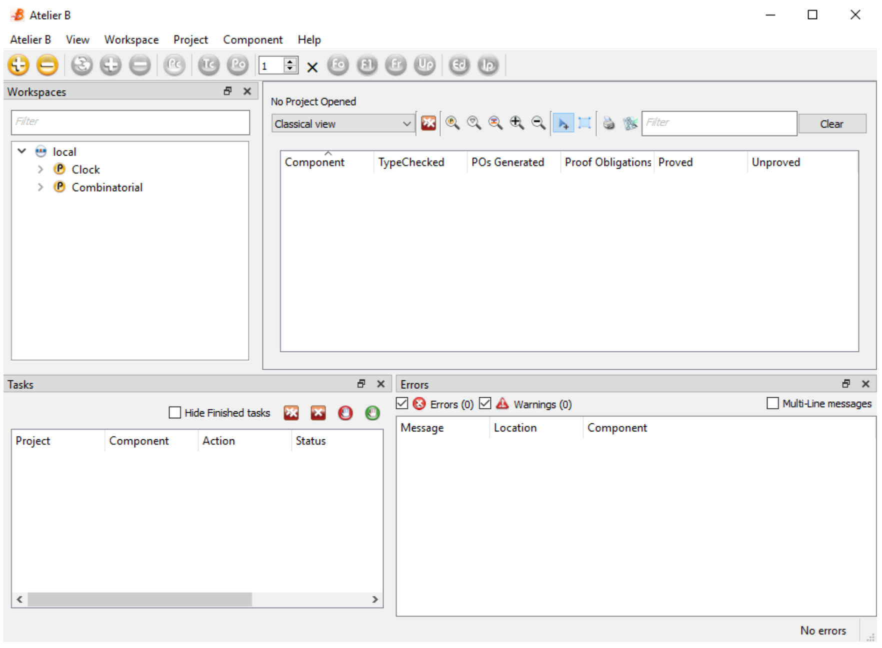

Atelier B
Atelier B View Workspace Project Component Help
Workspaces
Filter
local
Clock
Combinatorial
No Project Opened
Classical view
Filter
Clear
Component TypeChecked POs Generated Proof Obligations Proved Unproved
Tasks
Hide Finished tasks
Project Component Action Status
Errors
Errors (0) Warnings (0)
Multi-Line messages
Message Location Component
No errors

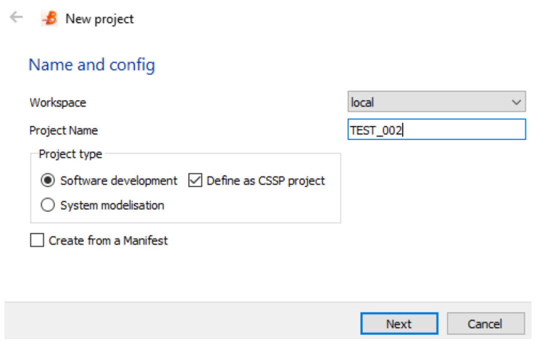
Figure 4.3: The Atelier CSSP main window when executing the startAB.cmd script

[1] When you buy a CSSP starter kit, you receive by email a link to download the IDE.

**Troubleshooting**
If the two projects *Clock* and *Combinatorial* do not show up on the project list pane (left), you probably do not have the rights to modify the Atelier B configuration files in the directory press/bdb.

Power the board by using the power supply. Connect the CLEARSY Safety Platform to your PC with a micro-USB cable. The board should now have some LEDs on after few seconds. The board is executing the program that was previously uploaded in memory. If it is the very first use of the board, the flash memory is empty and the board is literally doing nothing.
Let us create our first CLEARSY Safety Platform project:

- go to the menu Atelier B / New / Project.
- select "Software development" and "Define as CSSP project" (figure 4.4).
- enter a project name that is not yet defined in the project list pane

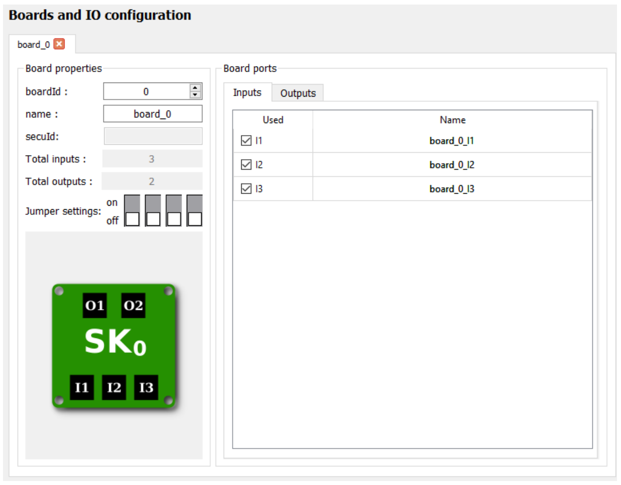


Name and config

Workspace: local
Project Name: TEST_002
Project type
[● Software development] [☑ Define as CSSP project]
○ System modelisation
☐ Create from a Manifest

Figure 4.4: Information required to create a CSSP project

- click "Next" then "Finish".
- a new window requires you to select the board type - keep the default choice "SK0" and press "OK".
- a new window shows up to configure the number and names of the IOs.
- click on "create new board".
- a graphical representation of the board appears together with two panes on the right that allow to select the inputs to use and to edit their default name (figure 4.5).



Boards and IO configuration

Board properties
boardId : 0
name : board_0
secuId:
Total inputs : 3
Total outputs : 2
Jumper settings: on / off

Board ports

| Used | Name |
|---|---|
| ☑ I1 | board_0_I1 |
| ☑ I2 | board_0_I2 |
| ☑ I3 | board_0_I3 |

Figure 4.5: The configuration window to select the inputs/outputs and to modify the default naming

- once your are happy with the configuration, click "Next".
- a summary page is displayed with the exact configuration of the board.
- click on "Finish" - you are asked to confirm the writing of the configuration. As we are creating the project, there is no previous configuration, so we can safely save it. Click on "Yes".
- the components of the project are generated in few seconds.
- then you finally get several orange boxes in the main window. The boxes represent the components generated from the configuration (IOs numbers and names). The colour represents their proof status: red/orange is "not proved", green is "proved" (figure 4.6).

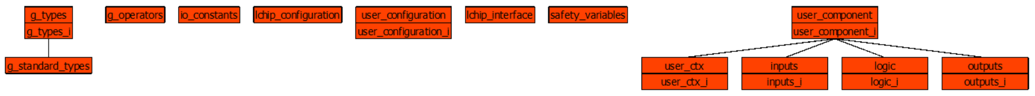


Figure 4.6: The CSSP project generated from the board configuration. Boxes are components.

- select all the components with Ctrl-A.
- initiate their proof with Ctrl-0 (or click on the blue button "F0" on the toolbar).
- within 30 seconds, all the components turn to green. The project is completely proved.
- now that the correctness of the project is ensured, we need to compile it. Right click on the project name (left pane) and select "CSSP Runner"[2].
- a new window - the CSSP runner - shows up, representing graphically the different stages of the compilation (figure 4.7).

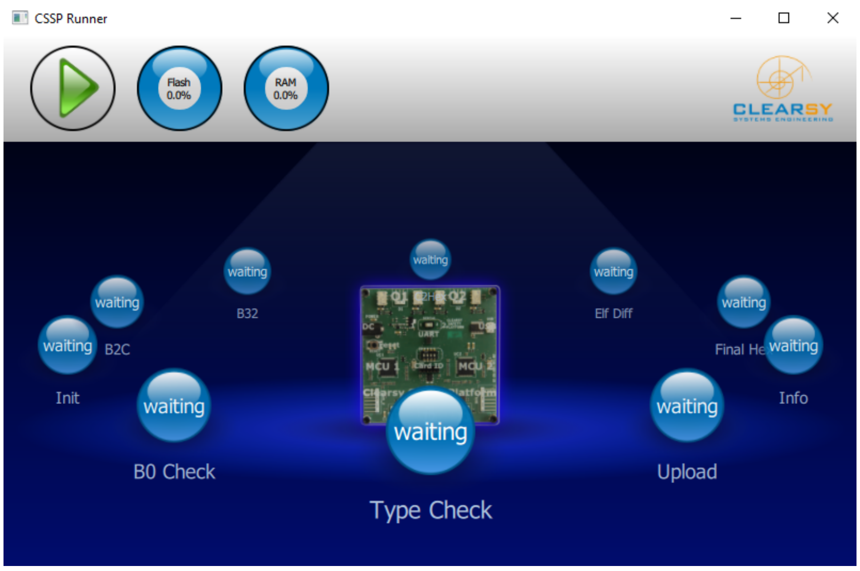


Figure 4.7: The CSSP runner shows the different stages of the compilation process

- click the green triangle on the top left to initiate the compilation
- an animation shows how the different stages are completed with (normally) a green tick for all of these. The memory footprint is displayed on the top during the penultimate stage.
- finally the runner reaches the last stage - upload. You are asked to reset the SK0 board. Push on the reset button on the board; the board starts blinking as it enters the bootload mode [3]

[2]there is also a "CSSP Runner SK1" to use only with the board SK1.
[3]This mode is used to modify the program in flash.

and downloads the program.
- once the transfer is completed, you are asked to reset the board again, to leave the bootload mode. The program in flash is copied in memory then its execution is initiated.
- the program does nothing (no modification of the outputs): so normally you should observe the two outputs O1 and O2 of the boards set to OFF (LEDs off), and the board status LED blinking slowly.

Congratulations! You have successfully programmed the SK0 board for the first time!

## 4.3 The programming model

The CLEARSY Safety Platform is aimed at **automation functions for cyclic applications**, as it
- reads the digital inputs.
- performs computations using a subset of the B language.
- modifies the outputs.
- reads the current time since the central unit started.

The function is executed regularly as often as possible similar to Arduino programming (setup(), loop()). There is no underlying operating system and no interrupt. There is no delay(). There

```
init();

while (1) {
  function$_{binary1}$();
  function$_{binary2}$();
}
```

Figure 4.8: The pseudo-code of the main loop

is no predefined cycle time. However if the outputs are not set and cross-read every 50 ms by microcontrollers, the board $SK_0$ enters panic mode and reboots.
Inputs are values captured at the beginning of a cycle (instantaneous values). Input capture is synchronised on both microcontrollers in order to acquire the same values (and prevent unwanted reboot because of different behaviour over the two central units). Outputs are maintained from one cycle to another.

With the CLEARSY Safety Platform, the developer is incited to focus on developing a function, independently of its transformation and distributed execution.

## 4.4 Development cycle: the steps

The CSSP development cycle strictly follows the B method which can be summarised as:
- specification model is written first, then comes the implementation model, both using the same language (B).
- models are proved.
- source code is generated from implementation models.

The whole picture (figure 4.9) contains more details that are going to be explained. Fortunately the CSSP IDE helps to lower the number of actions to perform when developing an application.

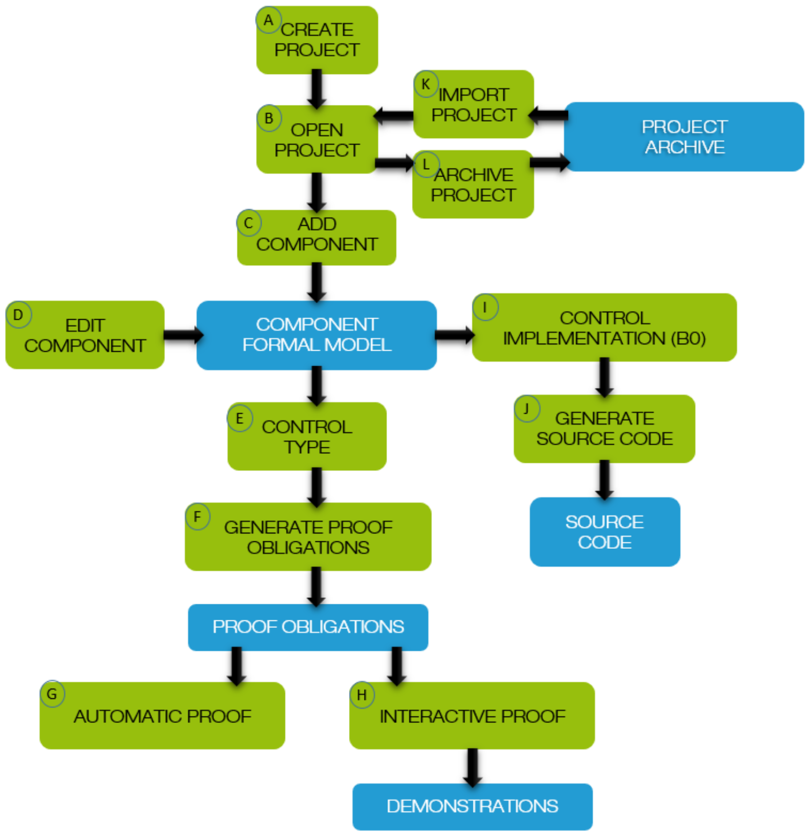


Figure 4.9: The development cycle with the B method - blue boxes are files, green ones are actions performed with the CSSP IDE.

### 4.4.1 CSSP project creation

Creating a CSSP project[4] covers actions (A), (B), and (C). The project is created and populated with the required components (§ 11.1) based on your board configuration (number and naming of inputs and outputs). There is no need to add other components for simple applications.

R It is not possible to regenerate a project once it has been created from a given configuration, in case for example you made a spelling mistake that you want to correct. As the components are automatically generated, you are going to loose all edits. Just create another project and copy/paste your edits from one project to the other.

### 4.4.2 Editing CSSP components

Editing CSSP components covers action (D). You are entitled to modify the following components:

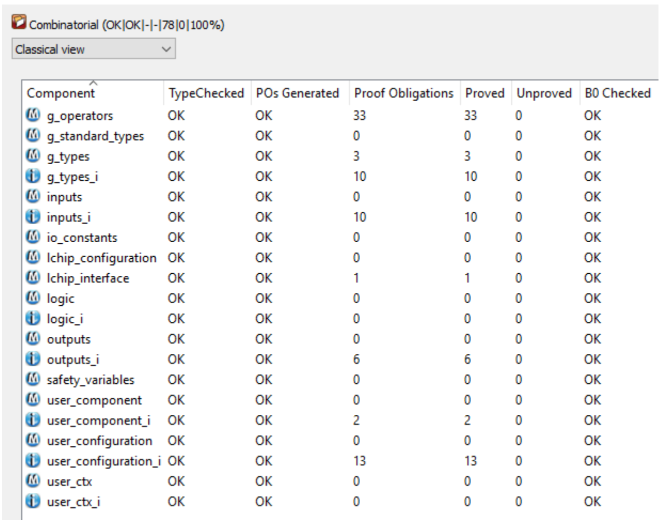
- **user_ctx** by
  - adding constants (clause CONCRETE_CONSTANTS) (optional),
  - providing properties defining constants' types and constraints (clause PROPERTIES),
  - adding sets (clause SETS) (optional).
- **user_ctx_i** by providing values to your concrete constants and sets (clause VALUES)
- **user_logic** by modifying the specification of the operation user_logic. By default, the specification is skip, meaning that all variables defined at specification level are not modified. The clauses INVARIANT and INITIALISATION could be modified as well.

[4] A software development project + CSSP option

- **user_logic_i** by
  - adding variables (clause CONCRETE_VARIABLES) (optional),
  - adding typing properties for variables (clause INVARIANT) ,
  - adding initialisation values for variables (clause INITIALISATION),
  - modifying the body of the operation user_logic (clause OPERATIONS).

### 4.4.3 Proving CSSP project

Proving CSSP project covers actions (E), (F), and (G). Initiating automatic proof by selecting one/several/all components and clicking on the "F0" button on the toolbar will start automatic proof in force 0 of these components. If required, type control and proof obligation generation are performed automatically. Your project is fully proved if all component boxes are green with graphical view or 100% proved with classical view (figure 4.10).



| Component | TypeChecked | POs Generated | Proof Obligations | Proved | Unproved | B0 Checked |
|---|---|---|---|---|---|---|
| g_operators | OK | OK | 33 | 33 | 0 | OK |
| g_standard_types | OK | OK | 0 | 0 | 0 | OK |
| g_types | OK | OK | 3 | 3 | 0 | OK |
| g_types_i | OK | OK | 10 | 10 | 0 | OK |
| inputs | OK | OK | 0 | 0 | 0 | OK |
| inputs_i | OK | OK | 10 | 10 | 0 | OK |
| io_constants | OK | OK | 0 | 0 | 0 | OK |
| lchip_configuration | OK | OK | 0 | 0 | 0 | OK |
| lchip_interface | OK | OK | 1 | 1 | 0 | OK |
| logic | OK | OK | 0 | 0 | 0 | OK |
| logic_i | OK | OK | 0 | 0 | 0 | OK |
| outputs | OK | OK | 0 | 0 | 0 | OK |
| outputs_i | OK | OK | 6 | 6 | 0 | OK |
| safety_variables | OK | OK | 0 | 0 | 0 | OK |
| user_component | OK | OK | 0 | 0 | 0 | OK |
| user_component_i | OK | OK | 2 | 2 | 0 | OK |
| user_configuration | OK | OK | 0 | 0 | 0 | OK |
| user_configuration_i | OK | OK | 13 | 13 | 0 | OK |
| user_ctx | OK | OK | 0 | 0 | 0 | OK |
| user_ctx_i | OK | OK | 0 | 0 | 0 | OK |

Figure 4.10: Proof status with classical view: 100% proved requires to have only zeros in the "Unproved" column.

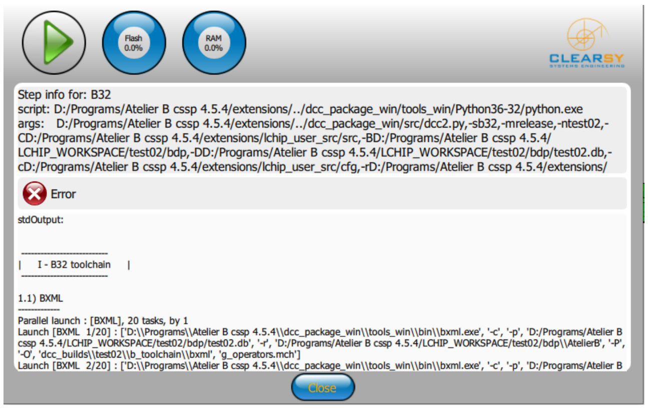
### 4.4.4 Interactively proving CSSP project

Interactively proving CSSP project covers action (H). A project not fully proved automatically does not necessarily mean that the project contains errors. Unproved proof obligations might be not "adequate"/too complicated for the proof tools and require some guidance from the developer to complete the mathematical demonstration like starting a proof by contradiction or by cases, rewriting/simplifying predicates/expressions, etc. Interactive proof is an advanced topic and as such will be addressed later.

### 4.4.5 Generating code for CSSP project

Generating code for CSSP project covers actions (I) and (J). The CSSP Runner is triggered by selecting the project, right-clicking and then by clicking "CSSP Runner" in the drop-down menu. A new window shows up with a rotating view of the different compilation steps. After clicking the green triangle button on the top left, the different stages are executed.

- If a stage fails, you get a red cross and the compilation process stops (figure 4.11). If you click on the "error" text, you get the logs captured during the execution of the related tool. These logs may be copied by right clicking and selecting the action "copy" from the drop-down menu.



```
Step info for: B32
script: D:/Programs/Atelier B cssp 4.5.4/extensions/../dcc_package_win/tools_win/Python36-32/python.exe
args:   D:/Programs/Atelier B cssp 4.5.4/extensions/../dcc_package_win/src/dcc2.py,-sb32,-mrelease,-ntest02,-
CD:/Programs/Atelier B cssp 4.5.4/extensions/lchip_user_src/src,-BD:/Programs/Atelier B cssp 4.5.4/
LCHIP_WORKSPACE/test02/bdp,-DD:/Programs/Atelier B cssp 4.5.4/LCHIP_WORKSPACE/test02/bdp/test02.db,-
cD:/Programs/Atelier B cssp 4.5.4/extensions/lchip_user_src/cfg,-rD:/Programs/Atelier B cssp 4.5.4/extensions/

stdOutput:


 ---------------------------
|    I - B32 toolchain     |
 ---------------------------

1.1) BXML
-------------
Parallel launch : [BXML], 20 tasks, by 1
Launch [BXML  1/20] : ['D:\\Programs\\Atelier B cssp 4.5.4\\dcc_package_win\\tools_win\\bin\\bxml.exe', '-c', '-p', 'D:/Programs/Atelier B
cssp 4.5.4/LCHIP_WORKSPACE/test02/bdp/test02.db', '-r', 'D:/Programs/Atelier B cssp 4.5.4/LCHIP_WORKSPACE/test02/bdp\\AtelierB', '-P',
'-O', 'dcc_builds\\test02\\b_toolchain\\bxml', 'g_operators.mch']
Launch [BXML  2/20] : ['D:\\Programs\\Atelier B cssp 4.5.4\\dcc_package_win\\tools_win\\bin\\bxml.exe', '-c', '-p', 'D:/Programs/Atelier B
```

Figure 4.11: Clicking on the Error displays the logs captured during the execution of the related tool.

- If all the compilation stages succeed, you get information about the memory consumption for both Flash and RAM. Check that you do not exceed 100%[5]. You are asked to reset the board to enter bootload mode. As soon as the reset button is pushed, the two upload LEDs start blinking for around 30 seconds. Then the last stage gets a green tick and you are asked to reset again the board to leave bootload mode. Once the reset button is pressed, the board starts executing the software copied in Flash memory within 2 seconds.

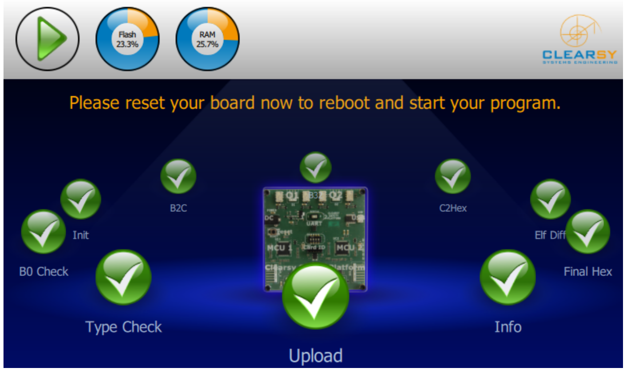


Figure 4.12: All green ticks indicate a successful compilation and upload.

### 4.4.6 Archiving and importing for CSSP project

Archiving and importing for CSSP project cover actions (K) and (L). To archive a project, select it, right-click and select "archive". Chose the destination directory (the archive file has the .arc extension). Chose "whole project" then "Next" and "Finish". To import a CSSP project, select "Workspace/restore project". Click "Next". Select the archive file to import, give a name to the project and click "Next" then "Finish". A new project appears in the left pane, with the contents of the archived project.

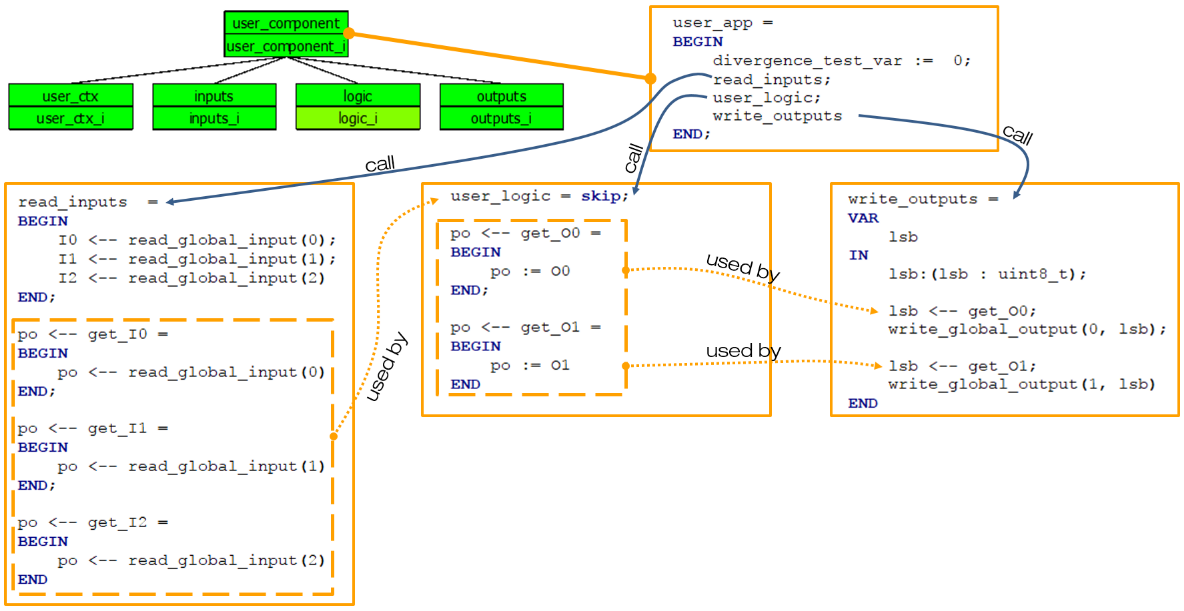
## 4.5 Programming the board

Programming the board (figure 4.13) consists of:

- describing, in the component *logic*, the behaviour of the board,

[5]In this case, you need to simplify your program to remain within this limit.

- declaring and using constants, defined in the component *user_ctx*,
- reusing operations defined in the component *inputs* and in the safety library.



```
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
```

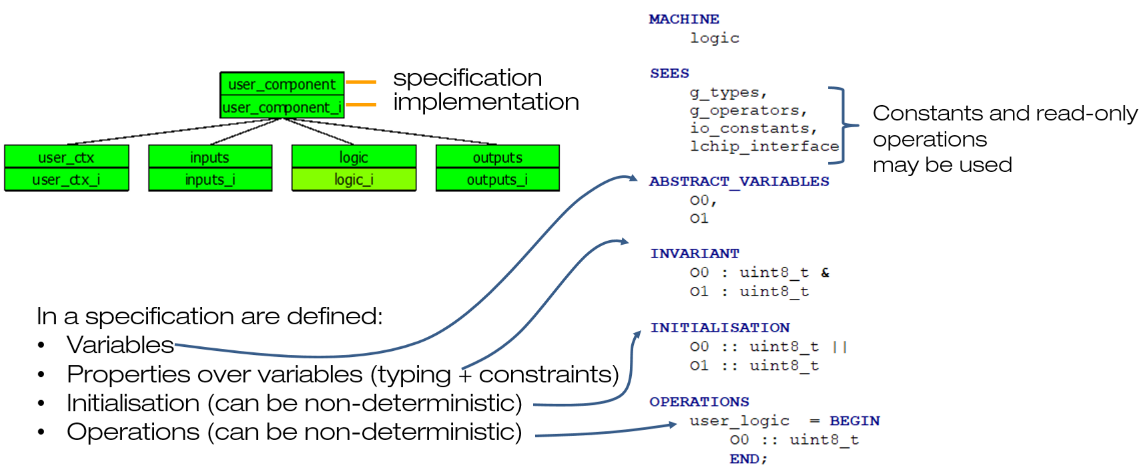
```
user_logic = skip;

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 4.13: The default CSSP project showing the dependencies between the components.

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*.
The structure of the component *logic* is seen in figure 4.14. It contains several clauses:

- SEES: contains the list of all seen components.
- ABSTRACT_VARIABLES: the outputs of the board. They are going to be refined in CONCRETE_VARIABLES (implementable) in the component *logic_i*.
- INVARIANT: the type of the variables, with some constraints between them (optional).
- INITIALISATION: the first value assigned to the variables. At specification level, initialisation may be non-deterministic (any value complying with some conditions).
- OPERATIONS: each operation specifies how the modelling variables are modified, preferably without the algorithmic details.



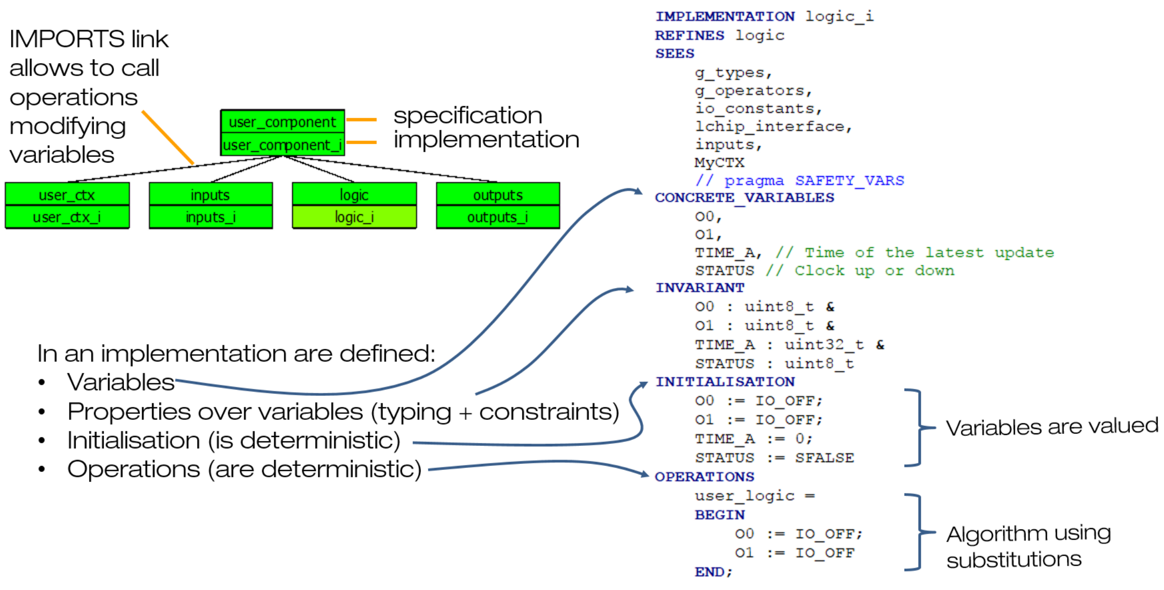
```
MACHINE
    logic

SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface

ABSTRACT_VARIABLES
    O0,
    O1

INVARIANT
    O0 : uint8_t &
    O1 : uint8_t

INITIALISATION
    O0 :: uint8_t ||
    O1 :: uint8_t

OPERATIONS
    user_logic  = BEGIN
        O0 :: uint8_t
    END;
```

In a specification are defined:
- Variables
- Properties over variables (typing + constraints)
- Initialisation (can be non-deterministic)
- Operations (can be non-deterministic)

Figure 4.14: The structure of the logic component (specification).

Once the specification component has been modified[6], it is time to modify the component *logic_i*. The structure of the component *logic_i* (figure 4.15) is quite similar to that of component *logic*:

- SEES: the implementation should see the same components.
- CONCRETE_VARIABLES: outputs variables have to be concrete (implementable). New variables (not defined in the component *logic*) could be added.
- INVARIANT: a concrete type (uint8_t, uint16_t, uint32_t, or BOOL) has to be given to all concrete variables declared. Board outputs must be declared as uint8_t.
- INITIALISATION: all variables have to be assigned a deterministic value. This value should comply with its type.
- OPERATIONS: the behaviour of each operation has to be implemented with only implementable substitutions.


IMPORTS link allows to call operations modifying variables
user_component
user_component_i
specification
implementation
user_ctx
user_ctx_i
inputs
inputs_i
logic
logic_i
outputs
outputs_i
In an implementation are defined:
• Variables
• Properties over variables (typing + constraints)
• Initialisation (is deterministic)
• Operations (are deterministic)
Variables are valued
Algorithm using substitutions

```
IMPLEMENTATION logic_i
REFINES logic
SEES
    g_types,
    g_operators,
    io_constants,
    lchip_interface,
    inputs,
    MyCTX
    // pragma SAFETY_VARS
CONCRETE_VARIABLES
    O0,
    O1,
    TIME_A, // Time of the latest update
    STATUS // Clock up or down
INVARIANT
    O0 : uint8_t &
    O1 : uint8_t &
    TIME_A : uint32_t &
    STATUS : uint8_t
INITIALISATION
    O0 := IO_OFF;
    O1 := IO_OFF;
    TIME_A := 0;
    STATUS := SFALSE
OPERATIONS
    user_logic =
    BEGIN
        O0 := IO_OFF;
        O1 := IO_OFF
    END;
```

Figure 4.15: The structure of the logic_i component (implementation).

## 4.5.1 Declaring variables

Figure 4.16 shows how variables are declared side-by-side in specification and implementation:

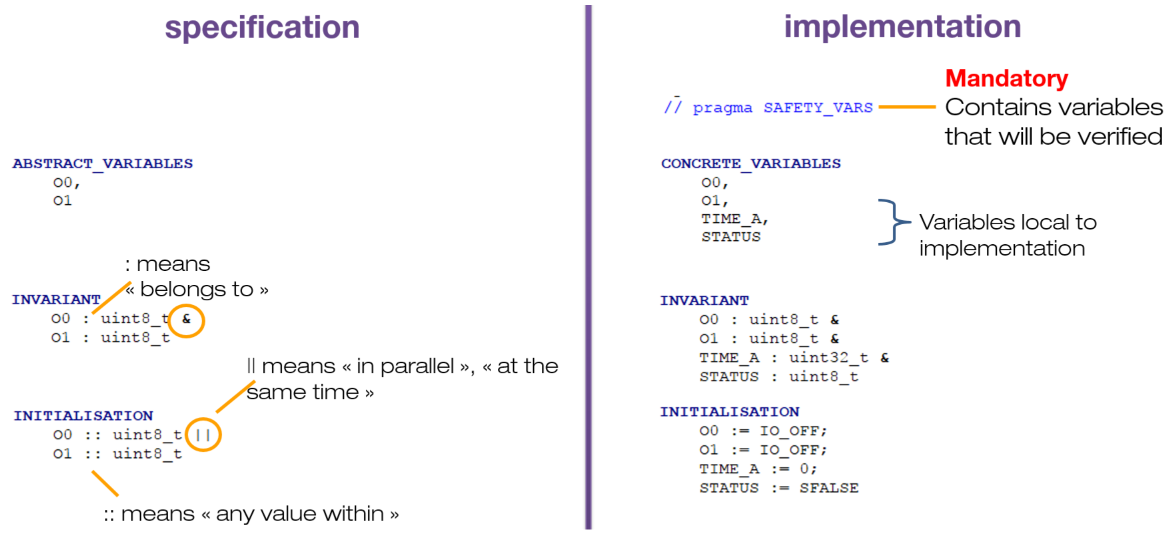
- first goes a list of variables, either abstract or concrete, separated by commas.
- in the implementation, the pragma SAFETY_VARS is mandatory to indicate that the implementation contains variables that need to be managed by the safety library.
- the invariant contains a list of predicates, separated by &, where types and constraints are expressed. Types in implementation are only implementable types.
- the initialisation describes the initial value of the variables. In the specification, the initialisation may be deterministic or non-deterministic. In the implementation, the initialisation has to be deterministic and to comply with the one in the specification[7]. Moreover initialisations in specification are performed "in parallel" by using the operator ||; in implementation, initialisations are ordered by using the operator ; (sequence).

## 4.5.2 Declaring constants

Figure 4.17 shows how constants are declared side-by-side in specification and implementation:

[6]It should mention *a minima* that the outputs are going to be modified when the operation user_logic is executed.

[7]If "vv belongs to uint16_t" was the initialisation of the variable vv, then "vv := 0" is a correct initialisation in implementation, while "vv := -1" is not.

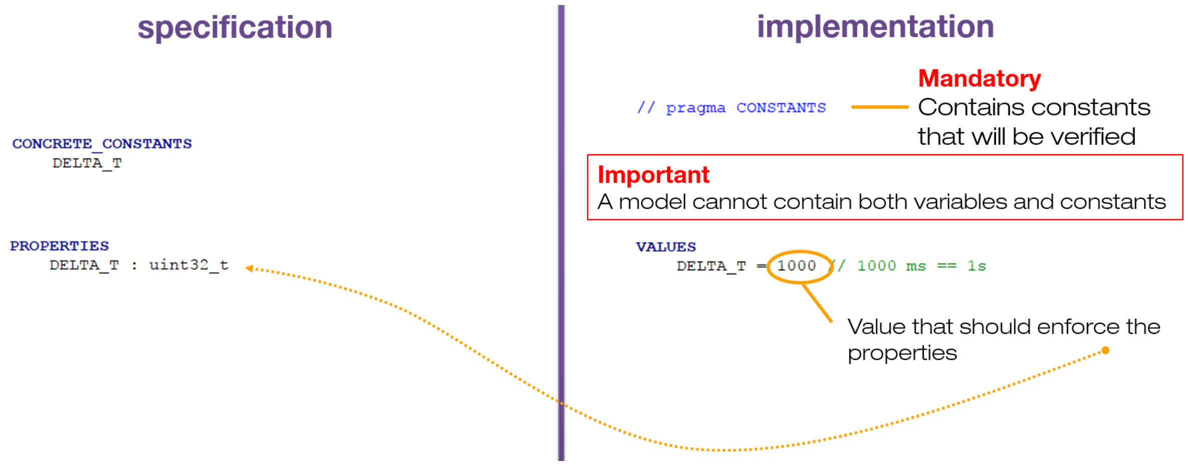

specification
implementation
Mandatory
Contains variables that will be verified
Variables local to implementation
: means « belongs to »
|| means « in parallel », « at the same time »
:: means « any value within »

```
ABSTRACT_VARIABLES
    o0,
    o1

INVARIANT
    o0 : uint8_t &
    o1 : uint8_t

INITIALISATION
    o0 :: uint8_t ||
    o1 :: uint8_t
```

```
// pragma SAFETY_VARS

CONCRETE_VARIABLES
    o0,
    o1,
    TIME_A,
    STATUS

INVARIANT
    o0 : uint8_t &
    o1 : uint8_t &
    TIME_A : uint32_t &
    STATUS : uint8_t

INITIALISATION
    o0 := IO_OFF;
    o1 := IO_OFF;
    TIME_A := 0;
    STATUS := SFALSE
```

Figure 4.16: Declaring variables in specification and implementation.

- first goes a list of constants, either abstract or concrete, separated by commas.
- in the implementation, the pragma CONSTANTS is mandatory to indicate that the implementation contains constants that need to be managed by the safety library.
- in the specification, the PROPERTIES contains a list of predicates, separated by &, where are expressed types and constraints. Properties are only in specification.
- in the implementation, the VALUES sets the values of the constants. Values provided have to comply with constant properties.


specification
implementation
Mandatory
Contains constants that will be verified
Important
A model cannot contain both variables and constants
Value that should enforce the properties

```
CONCRETE_CONSTANTS
    DELTA_T

PROPERTIES
    DELTA_T : uint32_t
```

```
// pragma CONSTANTS

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

Figure 4.17: Declaring constants in specification and implementation.

R An implementation should contain one and only one pragma: either SAFETY_VARS or CONSTANTS. That is why two components are made available: *user_ctx* for hosting constants and *logic* for variables and behaviour.

### 4.5.3 Describing a behaviour

Behaviour is described in OPERATIONS. Operations are populated with substitutions. Substitutions allow to describe how variables are (conditionally) modified. Available substitutions in specification are different from the ones available in implementation. We are going to see a number of these in what follows.

### Specification

In specification, substitutions express the properties that the variables comply with when the operation is completed independently from the algorithm implemented[8]. A good practice is to systematically use the substitution "becomes such that" (Figure 4.18).
A simple general form of the substitution is

$$var : (var : type(var))$$

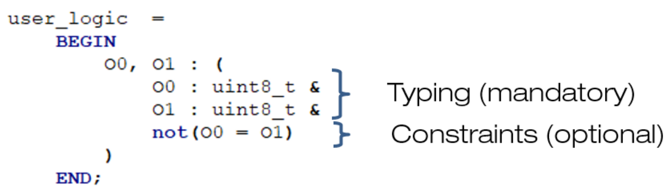
where *var* is an identifier and *type(var)* is a set expression compatible with the type of the variable *var*. It could also be of the form:

$$var\_list : (predicate\_list)$$

where *var_list* is a list of identifiers separated by commas and *predicate_list* is conjunction of predicates separated by &. Of course, if the value assigned to a variable is well-known, it is also possible to use the valuation substitution with:

$$var := value$$

where *value* is either a constant name, another variable identifier, a Boolean, or an Integer value. If none of the variables is going to be modified then use the substitution *skip*.



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

Figure 4.18: The substitution "becomes such that" used to specify that O0 and O1 are going to be modified within their type such that O0 is different from O1.

### Implementation

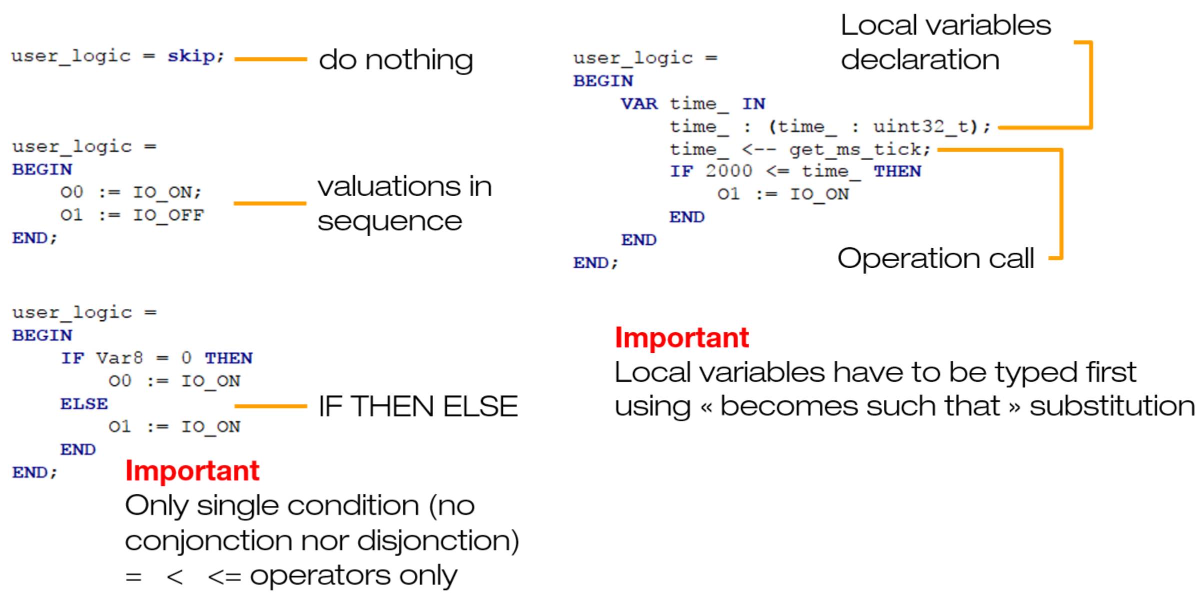
In implementation, several substitutions are available. Some of these are shown in figure 4.19.

- the *skip* substitution, when the variables are not modified.
- the assignment substitution *var := value*.
- the sequence substitution ; to order substitutions.
- the IF-THEN-ELSE substitution. Only a single condition is accepted. If several conditions have to be verified, several IF-THEN-ELSE substitutions have to be nested. Testing conditions have to be simple, hence computations have to be performed independently from (before) the test[9]
- the local variable substitution *VAR var IN substitutions END*. Local variables are used to store data and results of computation. Before any use of a local variable a substitution "becomes such that" has to be used to give a type to this local variable.
- the operation call substitution *var_list <– operation(var_list)* which use 0..n input parameters and 0..n return parameters.

These substitutions may be used in combination to obtain the final algorithm. The loop substitution will be introduced later on, in the projects.

[8]By using post-condition: the properties that are true when the substitution has been executed.

[9]For example, it is not possible to write IF xx+1<=yy THEN skip END. Instead a local variable has to be declared and used to compute xx+1. The test is then performed with this local variable.



```
user_logic = skip;
```

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

```
user_logic =
BEGIN
     IF Var8 = 0 THEN
          O0  := IO_ON
     ELSE
          O1  := IO_ON
     END
END;
```

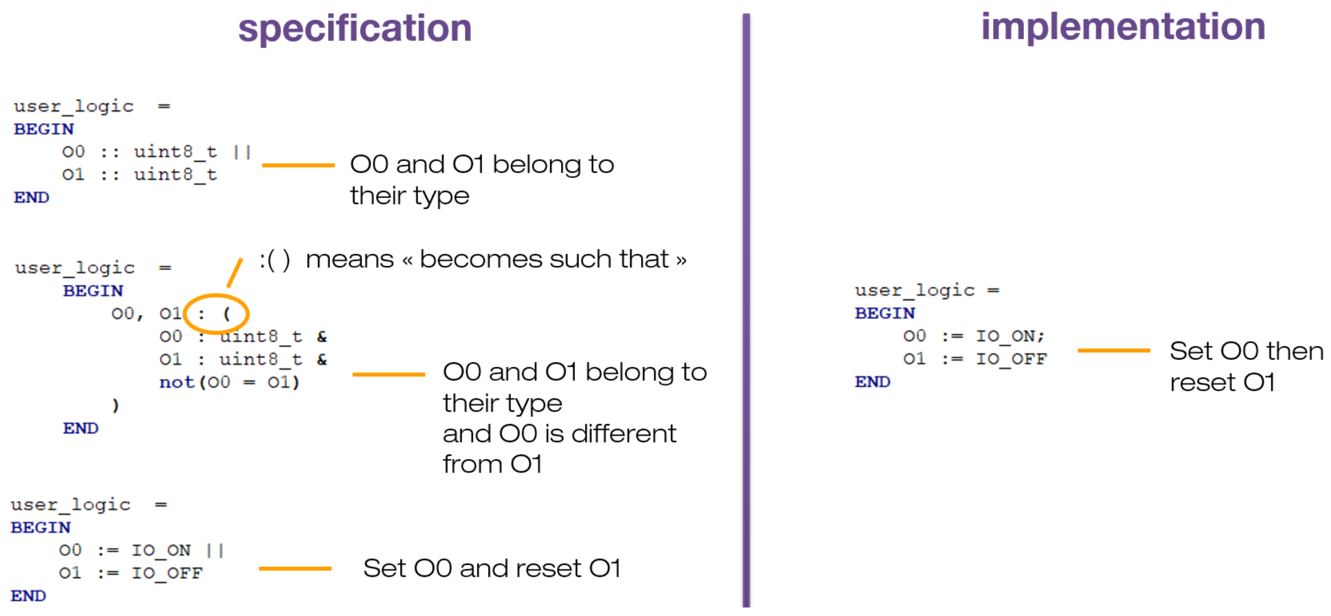
```
user_logic =
BEGIN
     VAR time_ IN
          time_  : (time_  : uint32_t);
          time_  <-- get_ms_tick;
          IF 2000 <= time_  THEN
               O1  := IO_ON
          END
     END
END;
```

do nothing
valuations in sequence
IF THEN ELSE
Important
Only single condition (no conjonction nor disjonction)
= < <= operators only
Local variables declaration
Operation call
Important
Local variables have to be typed first using « becomes such that » substitution

Figure 4.19: Some substitutions usable in implementation.

### Reconciling specification and implementation

If the writing of an implementation is quite straightforward, especially when the logic is not too complicated, the writing of a specification is far more complex. For example, the figure 4.20 shows



```
user_logic  =
BEGIN
     O0 :: uint8_t ||
     O1 :: uint8_t
END
```

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

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

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

specification
implementation
O0 and O1 belong to their type
:( ) means « becomes such that »
O0 and O1 belong to their type and O0 is different from O1
Set O0 and reset O1
Set O0 then reset O1

Figure 4.20: Several specification compatible with the implementation.

three different kinds of specification:

- the first one is fully non-deterministic. If one mistake is inserted in the implementation (both values at IO_OFF for example), it is not detected by the proof.
- the second one is non-deterministic but ensures that both values for $O_0$ and $O_1$ are different. Errors are detected except if the implementation is the contrary of what it should be.
- the last one is fully deterministic but very close to the implementation. One risk when having

such close specification/implementation is that we are only proving the copy/paste action between both models.

It is up to you to determine what kind of specification and more importantly what level of verification you are looking for. If your model only contains loosely constrained specification, the proof will not improve much the level of confidence in the algorithm.

# II Projects

# 5. Combinatorial

This first project introduces the combinatorial or asynchronous behaviour of the board. The outputs are updated as soon as the inputs are modified and not after a given period of time (a delay) which is a synchronous behaviour.
The equations we would like to implement are:

$$\mathrm{O}_0 = \mathrm{I}_1 \text{ and } \mathrm{I}_2 \text{ and } \mathrm{I}_3$$
$$\mathrm{O}_2 = \mathrm{not}(\mathrm{O}_1)$$

$\mathrm{O}_1$ will be ON only when all inputs are ON, unless it will be OFF (the AND function over three Boolean values). $\mathrm{O}_2$ is the opposite of $\mathrm{O}_1$: $\mathrm{O}_2$ is ON when $\mathrm{O}_1$ is OFF, and OFF when $\mathrm{O}_1$ is ON [1].

## 5.1 Modelling

The modelling requires three steps:

- The first step is to create a project, to give it a name and select "CSSP project".
- The second step is to add a board (only one) and to change the names of the inputs and outputs to respectively $I_1$, $I_2$, $I_3$ and $O_1$, $O_2$.
- the third step is to modify the components *logic* and *logic_i* to specify and implement the behaviour.

Listing 5.1: Non-deterministic specification of the operation user_logic

```
  user_logic =
  BEGIN
      O1 :: uint8_t ||
      O2 :: uint8_t
  END;
```

[1] This is a usual practice in the railways to detect if an equipment is failing or has no power supply. If both outputs are OFF, we know that the system is failing and the situation has to be handled with care.

For *logic*, we need to modify the specification of the operation user_logic. We could be precise and express directly the relationship between inputs and outputs, but for the very first experiment, we prefer to simply assert that the two outputs $O_1$ and $O_2$ are going to be modified non-deterministically[2] (see listing 11.14).

Listing 5.2: The implementation of the operation user_logic

```
 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
```

For *logic_i*, the variables 01 and 02 are already defined as CONCRETE
VARIABLES and both initialised with the value IO_OFF. No other variable is required for the implementation. We only need to modify the body of the operation user_logic (see listing 5.2).
We need three local variables to store the values of the three inputs $I_1$, $I_2$ and $I_3$. So we define three variables i1_, i2_ and i3_ with the substitution VAR ... IN ... END. The scope of this substitution in the END keyword, meaning that any use of these three variables outside of this scope will produce an error message. These three variables are typed prior to any use, by using the "becomes such that" substitution with the type uint8_t. These three typing substitutions have no effect on the modelling but are required by the compiler.
The values of the inputs are collected by calling the three operations get_I1, get_I2 and get_I3. The notation "var <– op" means that the operation op is called and returns one value that is assigned to the variable var. In our case, i1_, i2_ and i3_ are valued respectively with the values returned by get_I1, get_I2 and get_I3. The next 8 lines represent the computation of O1: O1 is first set to IO_OFF and then set to IO_ON only if the three inputs are all equal to IO_ON.

[2]The var :: type substitution is called non-deterministic because a particular value is provided. In the implementation, the value assigned to the variable var should belong to uint8_t. We remind you that the input and output digital values are represented by 8-bit values: IO_OFF and IO_ON.

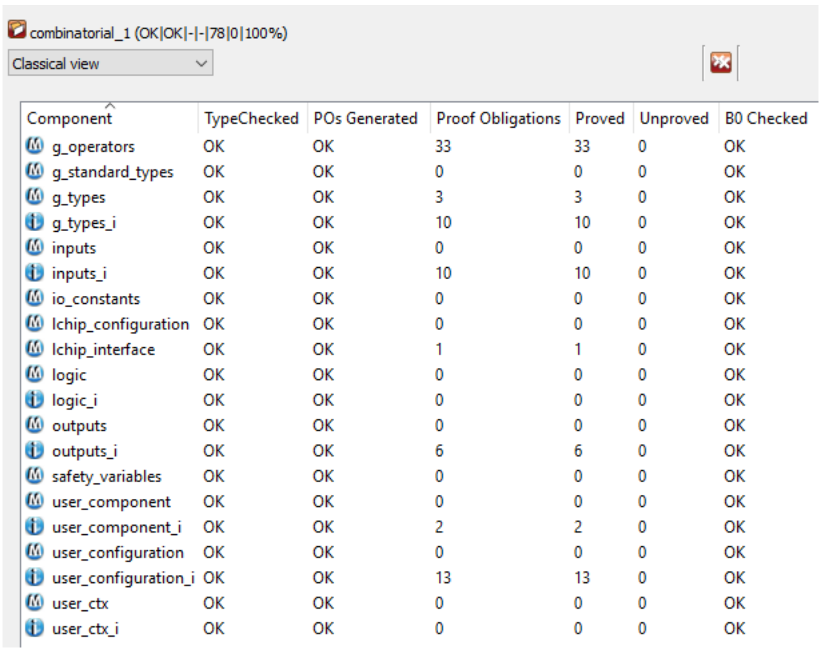
R The compiler in its current version does not support multiple testing conditions. Hence IF THEN ELSE have to be nested.

Finally O2 is computed with the last IF THEN ELSE, based on the value computed for O1.

R The sequence operator ; is not a line terminator like in C. Its role inside an operation[3] is to separate two substitutions. If extra ; are inserted in the model, error messages will be emitted when checking the model.

## 5.2 Executing

Now that the modelling is complete, we first need to check and prove the model. Select all the components by clicking on the central pane of the main window[4], then type ctrl+A. Type ctrl+0 to initiate the typecheck, proof obligation generation and proof of all these components. If some mistake was made, error messages are displayed either in the model editor or the error/warning pane in the main window. Be sure to correct any mistake before moving on. Finally press ctrl+U to complete the proof. You should obtain the proof status of the figure 5.1 (the Unproved column shows only zeros, meaning that the project is fully proved). The project is ready to be compiled.



| Component | TypeChecked | POs Generated | Proof Obligations | Proved | Unproved | B0 Checked |
|---|---|---|---|---|---|---|
| g_operators | OK | OK | 33 | 33 | 0 | OK |
| g_standard_types | OK | OK | 0 | 0 | 0 | OK |
| g_types | OK | OK | 3 | 3 | 0 | OK |
| g_types_i | OK | OK | 10 | 10 | 0 | OK |
| inputs | OK | OK | 0 | 0 | 0 | OK |
| inputs_i | OK | OK | 10 | 10 | 0 | OK |
| io_constants | OK | OK | 0 | 0 | 0 | OK |
| lchip_configuration | OK | OK | 0 | 0 | 0 | OK |
| lchip_interface | OK | OK | 1 | 1 | 0 | OK |
| logic | OK | OK | 0 | 0 | 0 | OK |
| logic_i | OK | OK | 0 | 0 | 0 | OK |
| outputs | OK | OK | 0 | 0 | 0 | OK |
| outputs_i | OK | OK | 6 | 6 | 0 | OK |
| safety_variables | OK | OK | 0 | 0 | 0 | OK |
| user_component | OK | OK | 0 | 0 | 0 | OK |
| user_component_i | OK | OK | 2 | 2 | 0 | OK |
| user_configuration | OK | OK | 0 | 0 | 0 | OK |
| user_configuration_i | OK | OK | 13 | 13 | 0 | OK |
| user_ctx | OK | OK | 0 | 0 | 0 | OK |
| user_ctx_i | OK | OK | 0 | 0 | 0 | OK |

Figure 5.1: Combinatorial_1 project status

Be sure that your SK0 board is connected with the USB connector to your host. Right-click on the project name in the left pane and select "CSSP Runner". A new window appears, showing a carousel of all the steps of the compilation. Click on the green triangle on the top left. All steps are cleared until the last one where you are asked to reset the SK0 board. Use a pen to push on the reset button and release it. The board starts to blink as it enters the bootload mode. After around 30s, the last step is cleared (green check) and you are again invited to reset the SK0 board. Push the reset button and release it. After 2 seconds, the board starts to execute your program.

[3]It is also used to separate operations
[4]the one showing all the components of the project.

## 5.3 Testing

To test your program, you need to be able to modify the status of the inputs in order to change the status of the outputs. You need three switches or three wires that you could plug/unplug to open/close the input circuits. The switches have to be connected to the two right pins as shown on figure 5.3. The status of the three inputs is indicated by three LEDs. One LED is ON when enough

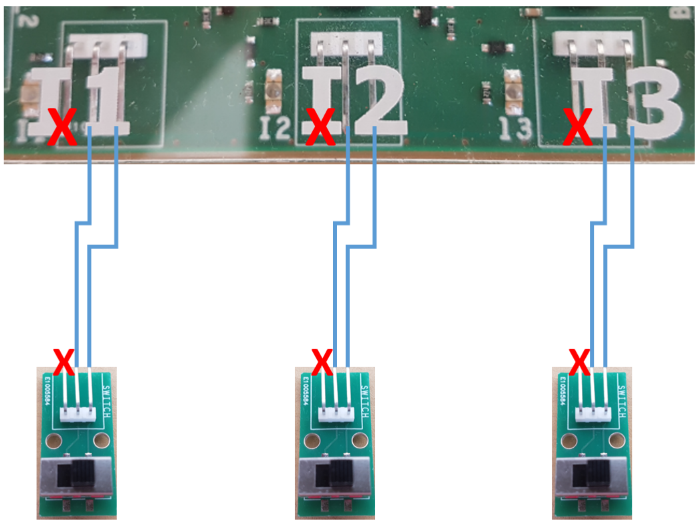


Figure 5.2: Connecting switches to the board

current is provided through the input circuit, OFF if not.
Connect your switches, change their status and see how the behaviour evolves. You should get the configurations shown in figure 5.3.

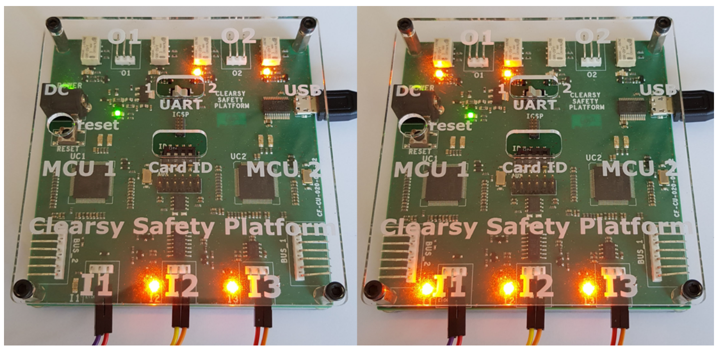


Figure 5.3: Two configurations: not all inputs are ON (left) and only O2 is ON; all inputs are ON (right) and only O1 is ON.

**Exercise 5.1** Create a program which implements the following equations:

$$O_0 = I_1 \text{ or } I_2 \text{ or } I_3$$
$$O_2 = \text{not}(O_1)$$

First create a new project and populate it with models similar to combinatorial_1. Then modify the body of the operation user_logic in the component *logic_i* to implement a disjunction over the values of the three inputs. ■

# 6. Clock

This second project introduces the synchronous behaviour of the board. The outputs are updated regularly after a given period of time (a delay).
The equations that we would like to implement are:

$$O_1 = \text{not}(O_1) \text{ every second (synchronous)}$$
$$O_2 = \text{not}(O_2) \text{ all the time (asynchronous)}$$

$O_1$ changes its status every second between OFF and ON. $0_2$ is the opposite of $O_1$: $0_2$ is ON when $O_1$ is OFF, and OFF when $O_1$ is ON.

## 6.1 Modelling

The modelling requires three steps:

- The first step is to create a project, to give it a name and select "CSSP project".
- The second step is to add a board (only one) and to change the names outputs to O1 and O2. Deselect the inputs as they are not going to be used in the following.
- The third step is to modify the components *logic*, *logic_i*, *user_ctx* and *user_ctx_i* to specify and implement the behaviour.

Listing 6.1: Non-deterministic specification of the operation user_logic

```
  user_logic =
  BEGIN
      O1, O2 :(
          O1 : uint8_t &
          O2 : uint8_t &
          not(O1=O2)
      )
  END;
```

For *logic*, we need to modify the specification of the operation user_logic. We simply assert that the two outputs O1 and O2 are going to be modified non-deterministically such as O1 and O2 are different (see listing 6.1).

Listing 6.2: Declaration of the constant delta_t in user_ctx

```
CONSTANTS
    delta_t
PROPERTIES
    delta_t: uint32_t &
    not(delta_t=0)
END
```

We need to modify *user_ctx* to introduce a constant representing the duration of the delay expressed in milliseconds. This constants is named delta_t; it is defined over 32 bits and is different from 0 (see listing 6.2).

Listing 6.3: Valuation of the constant delta_t in user_ctx_i

```
VALUES
    delta_t = 1000
END
```

In *user_ctx_i*, we provide a value for the constant deltat_t that is compatible with the two constraints: delta_t is a unsigned integer defined over 32 bits and is different from 0. We choose the value 1000 [1] (see listing 6.3).

Listing 6.4: Operation user_logic in user_ctx_i

```
    user_logic =
    BEGIN
        VAR ms_tick_cycle, s_tick_cycle, tick IN
            ms_tick_cycle :( ms_tick_cycle : uint32_t );
            s_tick_cycle :( s_tick_cycle : uint32_t );
            tick :( tick : uint32_t );

            ms_tick_cycle <-- get_ms_tick;
            s_tick_cycle := ms_tick_cycle / delta_t;
            tick := s_tick_cycle mod 2;

            IF tick = 0 THEN
                O1 := IO_ON
            ELSE
                O1 := IO_OFF
            END;
            IF O1 = IO_OFF THEN
                O2 := IO_ON
            ELSE
                O2 := IO_OFF
            END
        END
    END
```

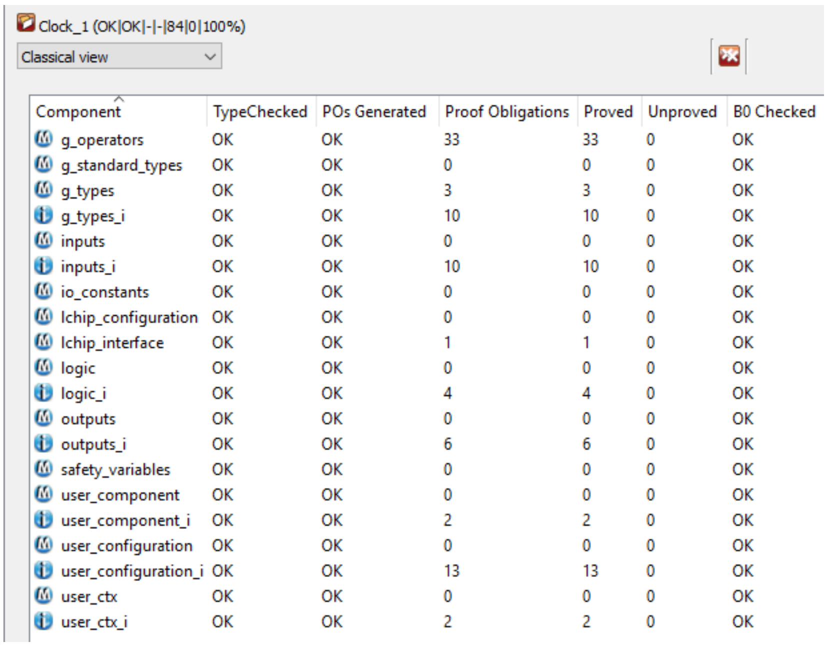
[1] Put for 1000 milliseconds.

For *logic_i*, We only need to modify the body of the operation user_logic (see listing 6.4). We need three local variables: ms_tick_cycle representing the number of milliseconds elapsed since the last reset, s_tick_cycle representing the number of times the delay has elapsed, and tick which represents the current state of the tick (OFF or ON). We define them with the substitution VAR ... IN ... END. These three variables are typed prior to any use, by using the "becomes such that" substitution with the type uint32_t.
The number of milliseconds elapsed since the last reset is collected by calling the operation get_ms_tick. The number of times the delay has elapsed is computed by divided the number of milliseconds elapsed since the last reset by the duration of the delay deltat_t. Finally the status of the tick (OFF or ON) is computed with a modulo 2 of the number of times the delay has elapsed. The output O1 is then set when tick is equal to 0 and reset when equals to 1. Finally O2 is computed with the last IF THEN ELSE, based on the value computed for O1.

R Do not set the delay deltat_t to a value less than 50 as very quick activation/deactivation of the output relays are going to kill them.

## 6.2 Executing

Now that the modelling is complete, we first need to check and prove the model. Select all the components by clicking on the central pane of the main window[2], then type ctrl+A. Type ctrl+0 to initiate the typecheck, proof obligation generation and proof of all these components. If some mistake was made, error messages are displayed either in the model editor or the error/warning pane in the main window. Be sure to correct any mistake before moving on. Finally press ctrl+U to complete the proof. You should obtain the proof status of the figure 6.1 (the Unproved column shows only zeros, meaning that the project is fully proved).


Clock_1 (OK|OK|-|-|84|0|100%)

Classical view

| Component | TypeChecked | POs Generated | Proof Obligations | Proved | Unproved | B0 Checked |
|---|---|---|---|---|---|---|
| g_operators | OK | OK | 33 | 33 | 0 | OK |
| g_standard_types | OK | OK | 0 | 0 | 0 | OK |
| g_types | OK | OK | 3 | 3 | 0 | OK |
| g_types_i | OK | OK | 10 | 10 | 0 | OK |
| inputs | OK | OK | 0 | 0 | 0 | OK |
| inputs_i | OK | OK | 10 | 10 | 0 | OK |
| io_constants | OK | OK | 0 | 0 | 0 | OK |
| lchip_configuration | OK | OK | 0 | 0 | 0 | OK |
| lchip_interface | OK | OK | 1 | 1 | 0 | OK |
| logic | OK | OK | 0 | 0 | 0 | OK |
| logic_i | OK | OK | 4 | 4 | 0 | OK |
| outputs | OK | OK | 0 | 0 | 0 | OK |
| outputs_i | OK | OK | 6 | 6 | 0 | OK |
| safety_variables | OK | OK | 0 | 0 | 0 | OK |
| user_component | OK | OK | 0 | 0 | 0 | OK |
| user_component_i | OK | OK | 2 | 2 | 0 | OK |
| user_configuration | OK | OK | 0 | 0 | 0 | OK |
| user_configuration_i | OK | OK | 13 | 13 | 0 | OK |
| user_ctx | OK | OK | 0 | 0 | 0 | OK |
| user_ctx_i | OK | OK | 2 | 2 | 0 | OK |

Figure 6.1: Clock_1 project status

The project is ready to be compiled. Be sure that your SK0 board is connected with the USB connector to your host. Right-click on the project name in the left pane and select "CSSP Runner".

[2]the one showing all the components of the project.

A new window appears, showing a carousel of all the steps of the compilation. Click on the green triangle on the top left. All steps are cleared until the last one where you are asked to reset the SK0 board. Use a pen to push on the reset button and release it. The board starts to blink as it enters bootload mode. After around 30s, the last step is cleared (green check) and you are again invited to reset the SK0 board. Push the reset button and release it. After 2 seconds, the board starts to execute your program.

## 6.3 Testing

To test your program, you just need to check that, when running, the behaviour is as expected: 0_1 beating every second and 0_2 is the opposite state of 0_2.

**Exercise 6.1** Create a program which implements the following equations:

$$O_1 = \text{not}(O_1) \text{ every second (synchronous)}$$
$$O_2 = \text{not}(O_2) \text{ every 2.5 seconds (synchronous)}$$

First create a new project and populate it with models similar to Clock_1. Add another constant delta_t1 valued with 2500. Then modify the body of the operation user_logic in the component *logic_i* to implement two clocks over the outputs 0_1 and 0_2 . ■

III

# Appendix

# 7. Hardware interface

The $SK_0$ board is a 10cm x 10 cm x 2 cm board with a weight of 130g. It offers several hardware interfaces to interact with that are listed and explained below.

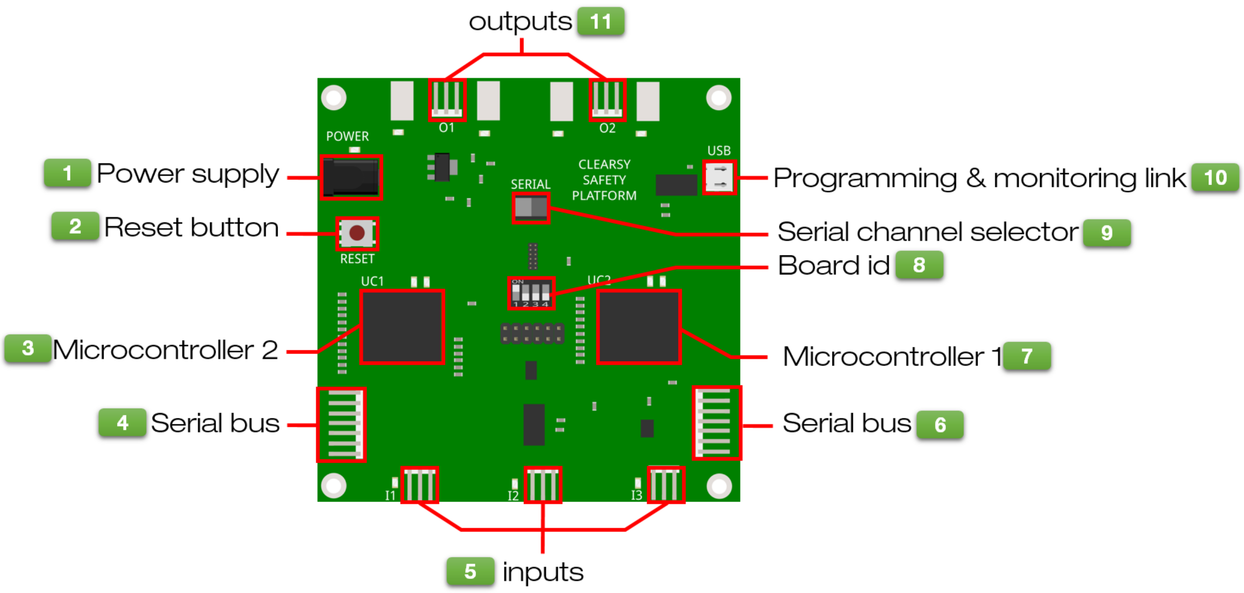


Figure 7.1: CLEARSY Safety Platform Starter Kit 0

## 7.1 Power supply

It requires +5V DC 500mA. This interface is privileged over USB port, as USB port implementation varies from one computer/equipment to another. We have encountered situations where not enough power was provided to the board, leading to erratic behaviour.

## 7.2 Reset button

The reset button resets the two microcontrollers at the same time. During the first two seconds after reset, if the bootloader receives a message from the USB port, it will enter bootload mode. If not, the program is copied from flash to RAM and starts its execution on both micro-controllers.

## 7.3 Microcontrollers 1 & 2

These are the two PIC32 micro-controllers [1] installed on the board. They both deliver around 100 DMIPS. As the function defined in *user_logic* is executed twice in sequence (binary$_1$ than binary$_2$), the SK$_0$ delivers around 50 DMIPS (not counting the execution time of the sequencer and the safety library).

## 7.4 Serial bus

The serial bus is used to connect several SK$_0$ together (see §10.1 for more information). If only one board is used, this bus is not functional and the board ID has to be 0b0000.

## 7.5 Inputs

There are 3 inputs on the board named I$_1$, I$_2$, and I$_3$.
One input (see figure 7.2) is made of 3 pins:

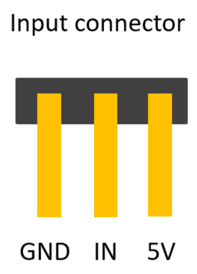


Figure 7.2: SK$_0$ input connector

- The ground (GND) is common to all inputs and outputs. In case the board is connected to another device (an Arduino for example), one of these GND pins has to be connected to the other device GND.
- The input pin (IN) on which is measured the input voltage.
- The +5V pin provides +5V. It has to be used when the input is a switch and not a power line. The switch has to be connected to both pins IN and +5V. This way, when the switch is closed, the current can flow from +5V to IN.

## 7.6 Board ID

The board ID is made of four bits b$_0$ b$_1$ b$_2$ b$_3$ (see figure 7.3), in order to discriminate the boards when connected together through the serial bus. There are some constraints when setting board IDs:

- If the board is used alone, its board ID should be 0b0000 (master board).
- If the board is connected to other SK$_0$ board(s) through the serial bus, all board IDs have to be different and one board should have the board ID 0b0000.

[1] PIC32MX795F512L with 512KB Flash and 128 KB RAM, delivering 105 DMIPS at 80MHz.

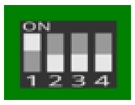
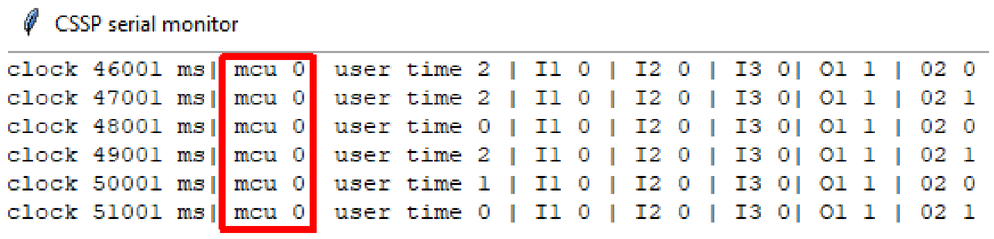


Figure 7.3: The board ID set to 0b0001.

If no board ID 0b0000 is present, the board(s) will not upload any new software, as the master board is supposed to initiate the upload process. From the user point of view, everything is working properly (from compilation to upload complete) but the program is not flashed in memory and the board is not updated. So always check for the board ID 0b0000 on your system.
Modifying the board ID during the execution of the program leads to $SK_0$ entering the panic mode.

## 7.7 Serial channel selector

This selector is used to select either micro-controller 1 or micro-controller 2 when monitoring the execution of the program. Changing the position of the selector requires reseting the board to be effective.



```
clock 46001 ms| mcu 0  user time 2 | I1 0 | I2 0 | I3 0| O1 1 | O2 0
clock 47001 ms| mcu 0  user time 2 | I1 0 | I2 0 | I3 0| O1 1 | O2 1
clock 48001 ms| mcu 0  user time 0 | I1 0 | I2 0 | I3 0| O1 1 | O2 0
clock 49001 ms| mcu 0  user time 2 | I1 0 | I2 0 | I3 0| O1 1 | O2 1
clock 50001 ms| mcu 0  user time 1 | I1 0 | I2 0 | I3 0| O1 1 | O2 0
clock 51001 ms| mcu 0  user time 0 | I1 0 | I2 0 | I3 0| O1 1 | O2 1
```

Figure 7.4: With the CSSP serial monitor, traces indicate that the micro-controller 1 (mcu 0) is being tracked.

## 7.8 Programming & monitoring link

This is a micro-USB interface for programming (flashing) the board and for monitoring its execution. It could also be used to power the board, but depending on the configuration (PC USB port, USB cable, etc.) behaviour could be unpredictable.

## 7.9 Outputs

There are 2 outputs on the board named $O_1$ and $O_2$.

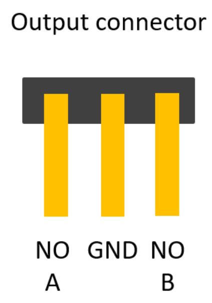


Figure 7.5: $SK_0$ output connector for both $O_1$ and $O_2$

One output (see figure 7.5) is made of 3 pins:

- The ground (GND) is common to all inputs and outputs. In case the board is connected to another device (an Arduino for example), one of these GND pins has to be connected to the other device GND.
- The "Normally Open" output pin A (NOA).
- The "Normally Open" output pin B (NOB).

An output is a switch, open or closed. When the switch is closed, the current can flow from the pin NOA to NOB (or from NOB to NOA, depending on how the board is connected to the outside world).

The outputs are not powered by the board. It is your responsibility to connect either NOA or NOB to a current source.

## 7.10 Electric constraints

The $SK_0$ board is not expected to directly power external devices. Except for low demand components, the board outputs are instead supposed to activate relays which in turn deliver power. Below are listed the constraints to maximise board durability:

- Input must be 5VDC from the 5VDC of the input pin or external power.
- Maximum output rating by output contact is:
  - 1 A at 30VDC and
  - 0.3 A at 125 VAC.
- The board is not protected against short circuit. Power consumption on inputs must be less than 100 mA.

# 8. LEDS on SK$_0$

The CLEARSY Safety Platform SK$_0$ is equipped with a number of LEDs providing indications of the board status. Their role is to:

- ensure fast checking that the board executes normally
- help identifying the root cause of unexpected behaviour

They are listed and explained below.

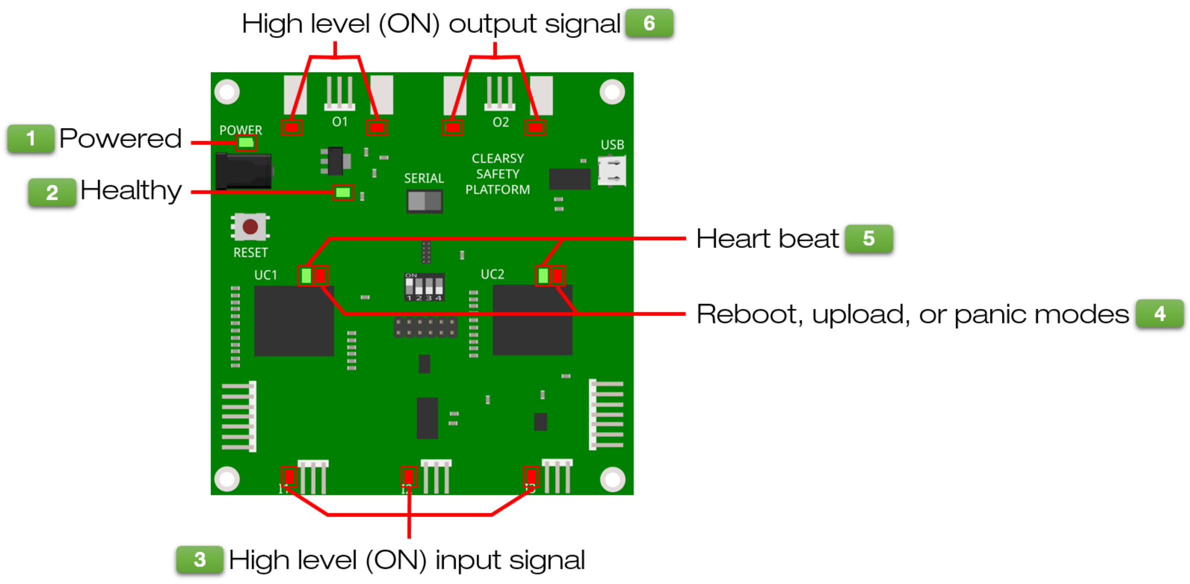

High level (ON) output signal 6
1 Powered
2 Healthy
POWER
O1
O2
CLEARSY SAFETY PLATFORM
SERIAL
USB
RESET
UC1
UC2
Heart beat 5
Reboot, upload, or panic modes 4
3 High level (ON) input signal

Figure 8.1: The LEDs installed on the SK$_0$

## 8.1 Powered

This green LED is ON when the board is powered with +5V. If the LED is not ON, check your power supply.

## 8.2 Healthy

This green LED is ON when both micro-controllers are powered with +3.3V. This voltage is issued from the main power supply.

## 8.3 High level input signal

Each red LED is ON when the electric level of the related input is ON. The input status can also be checked with the CSSP Serial Monitor (see §9). These indications on the electric level of inputs are not guaranteed if the board is only powered by the USB connector.

## 8.4 Reboot, bootload or panic modes

These two blinking (every second) red LEDs indicate that either the board is rebooting (the program in flash is being copied in memory) or is in bootload mode, erasing the previous program in flash with a new one. Pushing the reset button is required to enter / leave the bootload mode.
The two red LEDs blinking fast indicate a board in panic mode: the outputs are deactivated (circuits are open) and the board enters an infinite loop doing nothing.

## 8.5 Heartbeat

These two green LEDs blink synchronously every second when the board executes the program in flash memory and is healthy. *Heartbeat* and *Reboot, bootload or panic modes* LEDs are incompatible: either the former or the latter is ON and blinking.

## 8.6 High level output signal

Each of the two red LEDs is ON when the electric level of the related output is ON. The output status can also be checked with the CSSP Monitor. These indications of the electric level of outputs are not guaranteed if the board is only powered by the USB connector.

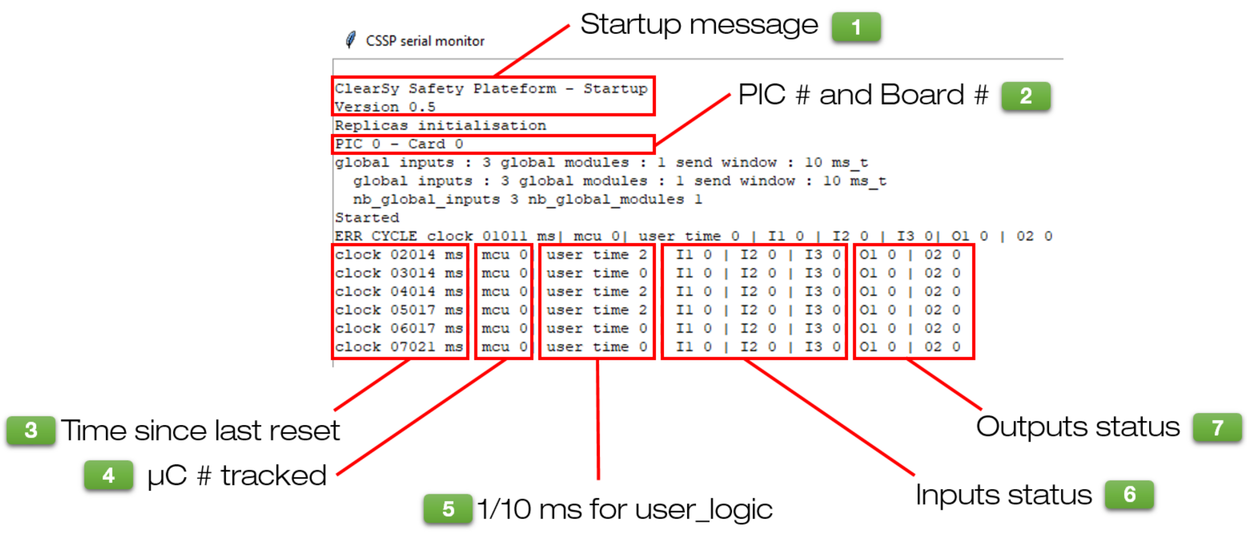
The outputs are not powered by the board. They are switches. The two red LEDs ON only indicate that the output circuit is closed.

# 9. CSSP Serial Monitor

The CSSP Serial Monitor is a feature offered at the project level. It allows to display the log messages emitted by the selected micro-controller on the serial bus.
It requires:

- one $SK_0$ board to be powered and connected through its the micro-USB port
- the CSSP runner not to be running. The USB communication medium is not shared.

To start the CSSP Serial Monitor, select the project, right-click then select "CSSP Monitor". A new window shows up, with 3 buttons (Quit, Pause, and Clear) and a central area containing the text messages emitted by the $SK_0$ board.



```
ClearSy Safety Plateform - Startup
Version 0.5
Replicas initialisation
PIC 0 - Card 0
global inputs : 3 global modules : 1 send window : 10 ms_t
  global inputs : 3 global modules : 1 send window : 10 ms_t
  nb_global_inputs 3 nb_global_modules 1
Started
ERR CYCLE clock 01011 ms| mcu 0| user time 0 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
clock 02014 ms| mcu 0| user time 2 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
clock 03014 ms| mcu 0| user time 0 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
clock 04014 ms| mcu 0| user time 2 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
clock 05017 ms| mcu 0| user time 2 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
clock 06017 ms| mcu 0| user time 0 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
clock 07021 ms| mcu 0| user time 0 | I1 0 | I2 0 | I3 0| O1 0 | O2 0
```

Figure 9.1: CSSP Serial Monitor messages

As soon as the board is connected and running, the messages are displayed in sequence every second, as shown in figure 9.1. To get all messages including those related to the board configuration, reset

the board while the serial monitor is executing.
Then you get the following information:

- A startup message, providing the version number of the software platform.
- The card number and the PIC number being tracked.
- The time elapsed since the last reset (in ms).
- The micro-controller being tracked (0 or 1).
- The time used for the execution of the user_logic operation during the last cycle (in tenths of ms).
- the inputs status (0: OFF, 1: ON).
- the outputs status (0: OFF, 1: ON).

# 10. Connecting several boards together

The board $SK_0$ is aimed at education and hence provides a limited number of inputs and outputs, in order to lower the production price and ease its dissemination. If the systems you are looking for require more inputs and/or outputs, you are advised to consider the board $SK_1$ with 20 inputs and 8 outputs.
However an "unsupported feature" allows you to connect several boards $SK_0$ together via their serial bus (see figure 10.1). All the boards are connected through a 7-pin connector (each pin is propagated at the same position to the next board). A configuration with *n* boards required *n-1* connectors. The last board has to be equipped with a terminator with two wires:

- UC1 RX and UC1 TX
- UC2 RX and UC2 TX

have to be connected together.

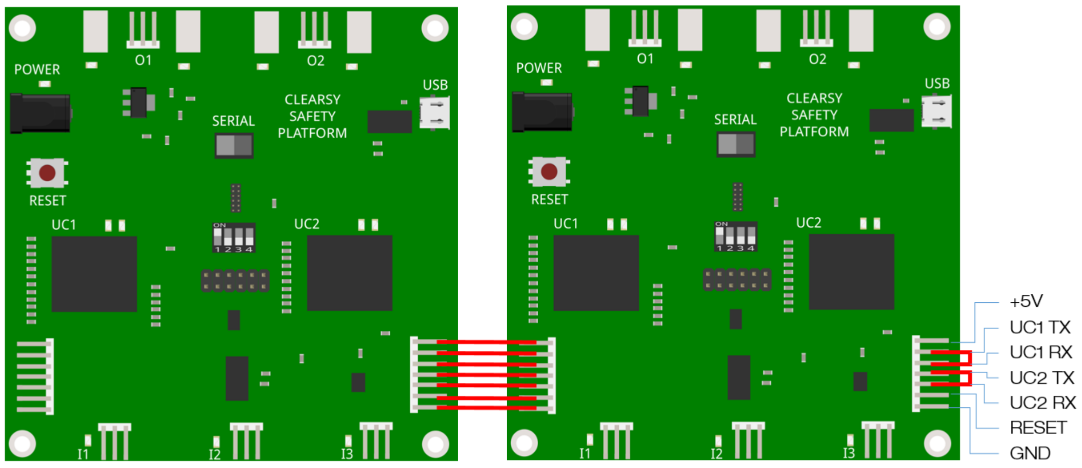


Figure 10.1: Connecting several $SK_0$ together

The boards execute exactly the same complete logic, making reference to all inputs and outputs.

Inputs acquired by one board are broadcast to other board through the serial bus, at the cost of degraded performance (cycle time is increased with the required data exchange between boards, the higher the number of boards, the longer the cycle time).

Board IDs all have to be different. One board has to have the ID 0 (the master board). The boards may be connected in any order. The terminator may be installed on any board.

This feature was developed before $SK_1$ was available. It was used to test the concept but many issues appeared during experiments including jittering distributed clocks. One solution adopted was to have one board initiating the communication (master) and providing a time reference for the other boards. This solution ensures more stability but breaks the safety principles (the master board may be faulty on the time and propagate this error to other boards without means of detection/correction).

**This multi-board configuration is provided without any support.**

# 11. Software interface

The software interface (see figure 11.1) is in two main parts:

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

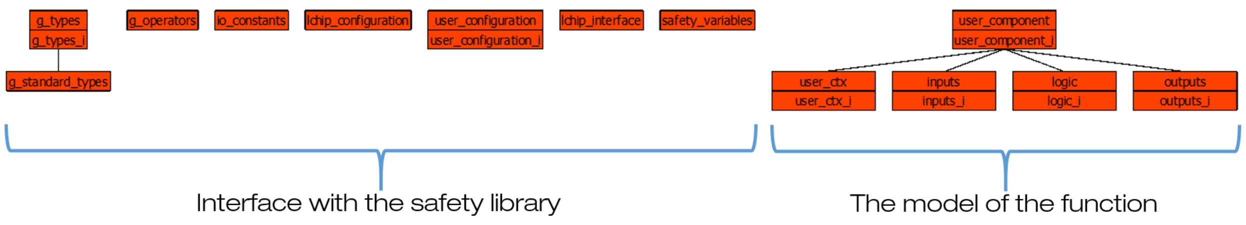


Figure 11.1: CSSP Software Interface

## 11.1 The interface with the safety library

### 11.1.1 g_types

Listing 11.1: Integer types defined in g_types

```
    uint32_t = 0..4294967295 &
    uint16_t = 0..65535 &
    uint8_t = 0..255 &
```

The component g_types defines the 3 integer types (listing 11.1) that must be used for implementing arithmetic computations on 8, 16 and 32 bits. No other integer type is available.

Components have to define a read-only access (clause SEES - listing 11.2) to the component g_types in order to be able to use any of these 3 types.

Listing 11.2: Clause SEES to insert in the referring component

```
SEES
    g_types
```

### 11.1.2 g_operators

6 bit-wise operators are defined (listing 11.3) for each supported type: uint8_t, uint16_t and uint32_t. These operators are shift-left (**sll**), shift-right (**srl**), negation (**not**), conjunction (**and**), disjunction (**or**) and exclusive disjunction (**xor**). They are defined as constant total functions:

- **sll** and **srl** have two parameters: the unsigned integer value to shift, the number of shifts to perform.
- **not** has one parameter: the unsigned integer value to negate.
- **and**, **or** and **xor** have two parameters: the unsigned integer values on which to perform logic operation.

Listing 11.3: 32-bit bit-wise operators defined in g_operators

```
    bitwise_sll_uint32 : uint32_t*uint8_t --> uint32_t &
    bitwise_srl_uint32 : uint32_t*uint8_t --> uint32_t &
    bitwise_not_uint32 : uint32_t --> uint32_t &
    bitwise_and_uint32 : uint32_t*uint32_t --> uint32_t &
    bitwise_xor_uint32 : uint32_t*uint32_t --> uint32_t &
    bitwise_or_uint32 : uint32_t*uint32_t --> uint32_t &
```

3 arithmetic operators are defined (listing 11.4) for each supported type: uint8_t, uint16_t and uint32_t. These operators are addition (**add**), subtraction (**sub**), and multiplication (**mul**).
They are defined as constant total functions with two parameters: the unsigned integer values on which to perform arithmetic operation.

Listing 11.4: 32-bit arithmetic operators defined in g_operators

```
    add_uint32 : uint32_t*uint32_t --> uint32_t &
    sub_uint32 : uint32_t*uint32_t --> uint32_t &
    mul_uint32 : uint32_t*uint32_t --> uint32_t &
```

These operators have been defined to ease the overflow proof[1]. With the CSSP, these operators are defined as a modulo (upper bound +1) of the result of the operation. For example

$$\text{add\_uint32}(x1, x2) = (x1+x2) \bmod (\text{MAX\_UINT32}+1)$$

where MAX_UINT32 is the upper bound of the type uint32_t. This way, the result always remains in the type of the function (8, 16 or 32 bits),the proof is thereby eased and made more automatic.

These logic and arithmetic constants have been implemented in the safety library. So they may be used directly in your implementation.

[1]With B, the result of an addition has to remain in its type. For example, the substitution val := 255+255 cannot be proved if val is defined as uint8_t as the result (512) exceeds the upper bound of the type. Usually this is solved by adding constraints on operands.

### 11.1.3 io_constants

This component defines two types (listing 11.5):

- TIME, defined over 32-bit integers, used to measure time (expressed in ms) with the operation get_ms_tick().
- IO_STATE, defined over 8-bit integers, contains the two valid states of the inputs and outputs: IO_OFF and IO_ON. The two values are chosen such that it is very unlikely that a memory perturbation produces the other valid value[2].

Listing 11.5: Constants defined in io_constants

```
ABSTRACT_CONSTANTS
        TIME,
        IO_STATE
CONCRETE_CONSTANTS
        IO_ON,
        IO_OFF
PROPERTIES
        TIME = uint32_t &
        IO_STATE = uint8_t &
```

### 11.1.4 lchip_configuration

This component defines three constants (listing 11.6) representing the maximum number of modules, inputs and outputs. This component is mainly aimed at easing the generation of source code and is of little interest for the developer.

Listing 11.6: Constants defined in lchip_configuration

```
        MAX_NB_MODULES : uint8_t &
        MAX_NB_INPUTS : uint8_t &
        MAX_NB_OUTPUTS : uint8_t &
```

### 11.1.5 lchip_interface

This component defines several functions:

- get_ms_tick, which returns the number of milliseconds elapsed since the last rest/power-on (listing 11.7),
- read_global_input, used to read the status of the digital inputs (used by the component inputs),
- write_global_output, used to modify the status of the digital outputs (used by the component outputs).

Listing 11.7: Constants defined in lchip_interface

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

[2] Setting an output with a value different from IO_OFF or IO_ON leads the board to enter panic mode.

### 11.1.6 user_configuration

This component defines several constants (listing 11.8) representing the number of modules, inputs, outputs, and their configuration (IDs). This component is mainly aimed at easing the generation of source code and is of little interest for the developer.

Listing 11.8: Constants defined in user_configuration

```
CONCRETE_CONSTANTS
    NB_MODULES,
    NB_INPUTS,
    NB_OUTPUTS,

    module_secu_ids ,
    module_nb_inputs ,
    module_nb_outputs ,

    input_module_ids ,
    input_local_ids ,

    output_module_ids ,
    output_local_ids
```

## 11.2 The model of the function

The model of the function to program contains 5 components; only two of these may be modified:

- user_ctx (and its implementation user_ctx_i), which contains only constants,
- logic (and its implementation logic_i), which contains only variables and operations.

### 11.2.1 user_component

This component contains the top-level function, user_app (listing 11.9), in charge of reading inputs (operation read_inputs), performing computation (operation user_logic) and modifying outputs (operation write_outputs). **This component should not be modified.**

Listing 11.9: The top-level operation user_app

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

### 11.2.2 user_ctx

This component contains the constants defined for the function to program.

The specification component, user_ctx (listing 11.10), has to declare the constants (clause CONCRETE_CONSTANTS) and their properties (clause PROPERTIES).

Listing 11.10: The DELTA_T constant from the project Clock

```
CONCRETE_CONSTANTS
        DELTA_T
PROPERTIES
        DELTA_T  :  uint32_t
```

The implementation component, user_ctx_i (listing 11.11), has to provide values (clause VALUES) to the constants defined in the component user_ctx.

Listing 11.11: Value for the DELTA_T constant from the project Clock

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

### 11.2.3 inputs

**This component should not be modified.**

It contains:

- the variables containing the status of the digital inputs (naming defined by the developer during the creation of the project). They are all defined as 8-bit integers with values in {IO_OFF, IO_ON}. These variables cannot be modified by other components, only updated when calling the operation read_inputs.

Listing 11.12: The declaration of the input variables

```
        board_0_I1  :  uint8_t  &
                board_0_I2  :  uint8_t  &
                board_0_I3  :  uint8_t
```

- the operations get_board_ (the ending depends on the names of the variables) return the status of each digital input, as read by the operation read_inputs. These operations are called by the operation user_logic (component logic_i).

Listing 11.13: The operations get_board_

```
        po <-- get_board_0_I1 =
                BEGIN
                                po := board_0_I1
                END;

                po <-- get_board_0_I2 =
                BEGIN
                                po := board_0_I2
                END;

                po <-- get_board_0_I3 =
                BEGIN
                                po := board_0_I3
                END
```

- the operation read_inputs, modifying the input status variables with the latest physical status read by the board (this operation is called by the top-level operation user_app).

Listing 11.14: Constants defined in g_operators

```
  read_inputs =
  BEGIN
       board_0_I1 <-- read_global_input(0);
                 board_0_I2 <-- read_global_input(1);
                 board_0_I3 <-- read_global_input(2)
  END;
```

### 11.2.4 logic

This component has to be modified by the developer:

- modify the specification of the operation user_logic (clause OPERATIONS) in the specification model logic.mch (listing 11.15).

Listing 11.15: Variables and operation user_logic specified in logic.mch

```
ABSTRACT_VARIABLES
      board_0_O1 ,
           board_0_O2
INVARIANT
      board_0_O1 : uint8_t &
           board_0_O2 : uint8_t
INITIALISATION
      board_0_O1 :: uint8_t ||
           board_0_O2 :: uint8_t
OPERATIONS
      user_logic = skip;
```

- if required declare additional variables in the implementation model logic_i.imp (clause CONCRETE_VARIABLES), then add a type (clause INVARIANT) and an initialisation (clause INITIALISATION) for each of these variables.
- modify the implementation of the operation user_logic (clause OPERATIONS) in the implementation model logic_i.imp (listing 11.16).

Listing 11.16: Variables and operation user_logic implemented in logic_i.imp

```
CONCRETE_VARIABLES
      board_0_O1 ,
           board_0_O2
INVARIANT
      board_0_O1 : uint8_t &
           board_0_O2 : uint8_t
INITIALISATION
      board_0_O1 := IO_OFF;
           board_0_O2 := IO_OFF
OPERATIONS
      user_logic = skip;
```

This component also contains operations (listing 11.17) to get access to the output status variables, named get_board_*, that are generated automatically from the board configuration (naming).
**The operations get_board_ should not be modified.**

Listing 11.17: Constants defined in g_operators

```
    po <-- get_board_0_O1 =
         BEGIN
                   po := board_0_O1
         END;

         po <-- get_board_0_O2 =
         BEGIN
                   po := board_0_O2
         END
```

### 11.2.5 outputs

**This component should not be modified.**

It contains the operation write_outputs (listing 11.18) that modifies the physical output status with the current output status variables read by the operations get_board_ (this operation is called by the top-level operation user_app).

Listing 11.18: The write_outputs operation defined in outputs

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

         lsb <-- get_board_0_O1;
                   write_global_output(0, lsb);

                   lsb <-- get_board_0_O2;
                   write_global_output(1, lsb)
    END
```

# 12. Troubleshooting

The CLEARSY Safety Platform combines several technologies which are constrained to fit the safety requirements. Most errors are linked to these restrictions (not all the B0 language is supported in implementation; additional information is required).
Some common sources of error:

- Using in implementation non-supported operators: +, -, and *. These operators are likely to generate overflow. The integer division / does not generate overflow and as such can be used.
- Using in implementation non-supported types: INT, NAT, and STRING.
- Writing a 32-bit unsigned int into a 16-bit or 8-bit.
- Writing a 16-bit unsigned int into a 8-bit.
- Allocating too much memory: table 49k of uint8_t == 100% memory
- Too many computations preventing inter-MCU verifications (browsing 29k cells of a table)
- Panic mode is obtained when there is a memory problem, the program transfer from host to SK0 was interrupted (CRC error) or when the board is unable to perform MCU verification on time.

Most error messages linked to models appear in the model editor (typecheck, compliancy with implementable language) or in the bottom pane (Errors & warnings) of the main window. Some more insidious errors located in the double compilation chain (DCC) may appear:

- in the runner window
- or in the dcc_build/log directory. For each compilation, one log file is generated (project name, date, time). For this file, search for error messages at the end of the file like "bxml error".

## 12.1 Upload fails immediately

During the compilation process, everything goes well except the last step which fails immediately. If the board is indeed connected with a USB cable able to transmit information (and not only provide electricity), then it is probably due to the USB Serial Port management by Windows. The $SK_0$ board requires such a port to ensure communication with the PC used for development. The USB serial ports are allocated by Windows when you connect a board on USB but sometimes

Windows fails to release the port; another serial port is then used, chosen among the one's available from COM2 to COM10. If COM11 is reached, it is not possible to set up communication between the board and the PC. To check if it is your case, open the Device Manager application and have a look at the *Ports (COM & LPT)* section (Fig. 12.1). If it shows COM11 or greater, it is not possible to establish a communication between the PC and the board.

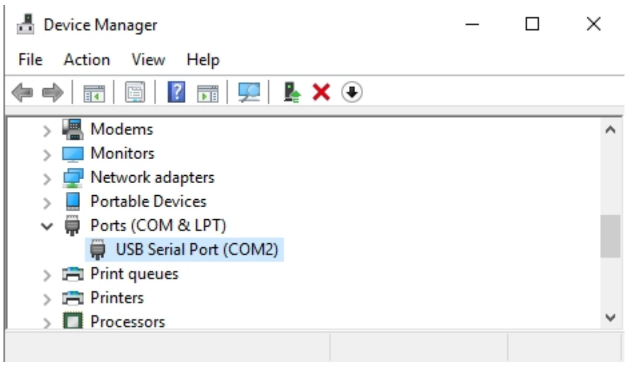
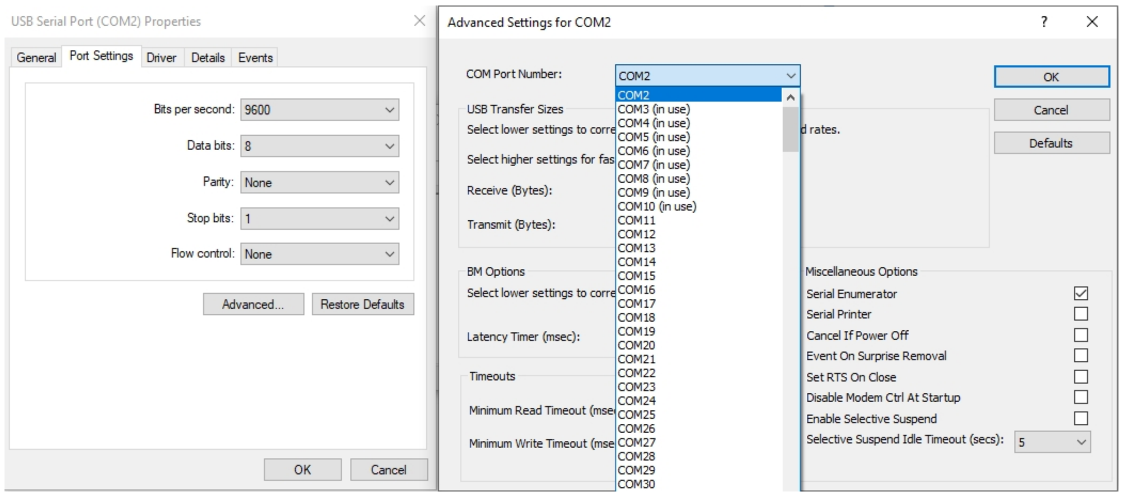


Figure 12.1: USB serial ports from the Device Manager

In this case, you need to reuse one port already allocated (*in use*). Right-click on the *USB Serial Port* item, then select *Properties*. A new windows shows up: switch to *Port Settings*, then click on the button *Advanced*. The window *Advanced Settings* shows up; the first field is *COM Port Number*: it indicates the current port used by the board. You need to select another one from COM2 to COM10 (shown as *in use*). Select it and confirm that you want to use this COM port. Close the windows and try to upload again the software on the board. It should work now.



Bits per second: 9600
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

COM Port Number: COM2

COM2
COM3 (in use)
COM4 (in use)
COM5 (in use)
COM6 (in use)
COM7 (in use)
COM8 (in use)
COM9 (in use)
COM10 (in use)
COM11
COM12
COM13
COM14
COM15
COM16
COM17
COM18
COM19
COM20
COM21
COM22
COM23
COM24
COM25
COM26
COM27
COM28
COM29
COM30

Select lower settings to corre
Select higher settings for fas
d rates.
Receive (Bytes):
Transmit (Bytes):
Select lower settings to corre
Latency Timer (msec):
Minimum Read Timeout (mse
Minimum Write Timeout (mse

Serial Enumerator ☑
Serial Printer ☐
Cancel If Power Off ☐
Event On Surprise Removal ☐
Set RTS On Close ☐
Disable Modem Ctrl At Startup ☐
Enable Selective Suspend ☐
Selective Suspend Idle Timeout (secs): 5

Figure 12.2: USB serial port advanced settings

# Glossary

**Vocabulary 12.1 — Atelier B.** CASE tool implementing the B method
**Vocabulary 12.2 — CSSP.** Abbreviation for CLEARSY Safety Platform
**Vocabulary 12.3 — Proof Obligation.** Mathematical predicate that needs to be demonstrated to prove a model. Non trivial models are made of hundreds/thousands of proof obligations.
**Vocabulary 12.4 — SIL{3/4}.** Safety Integrity Level. Defines the level of safety (the higher the safer) of a system - ranging between 0 and 4. Also refers to the level of danger of a system: a SIL4 system is considered to be able to kill people in case of catastrophic failure while a SIL0 system has no chance to harm someone.
**Vocabulary 12.5 — 2oo2.** Put for "2 out of 2". Means that a computation performed twice with different means is successful only if the two computations are identical. 2oo3 allows to have a system operated in the case that one of its three computers is behaving differently from the two others.

# Symbols Table

This appendix contains the description of reserved keywords and of the operators of B language, sorted by ascending ASCII order.
For each reserved keyword or operator, this chapter provides:

- its **ASCII notation**,
- its **mathematical notation**, if it differs from its ASCII notation.
- its **priority level**. The priority level corresponds to the priority level during the syntactic analysis. The higher the priority level of an operator, the more it attracts operands. For example, if the operators op40 and op250 are respectively of 40 and 250 priority, then the expression x op40 y op250 z is analysed as x op40 (y op250 z)
- its **associative properties** (L for associative to the left or R for associative to the right). If two binary operators named op have the same priority, then: x op y op z will be analysed as (x op y) op z if op is associative to the left; and as x op (y op z) if op is associative to the right.
- its **description**.

# B Language Keywords and Operators version 1.8.9

| ASCII | Math. | Pri. | As. | Description |
|---|---|---|---|---|
| ! | $\forall$ | 250 | | For any |
| " | | | | String or definition file delimiter |
| # | $\exists$ | 250 | | There exists |
| $0 | | | | Value of data before substitution |
| % | $\lambda$ | 250 | | Lambda expression |
| & | $\wedge$ | 40 | L | Conjunction (logical AND) |
| ' | | 250 | L | Access to a record field |
| ( | | | | Open bracket |
| ) | | | | Close bracket |
| * | $\times$ | 190 | L | Multiplication or Cartesian product |
| *x* ** *y* | $x^y$ | 200 | R | Power of |
| + | | 180 | L | Addition |
| +-> | $\nrightarrow$ | 125 | L | Partial function |
| +->> | ⤀ | 125 | L | Partial surjection |
| , | | 115 | L | Comma |
| - | | 180 | L | Subtraction |
| - | | 210 | | Unary minus |
| --> | $\rightarrow$ | 125 | L | Total function |
| -->> | $\twoheadrightarrow$ | 125 | L | Surjection |
| -> | $\rightarrow$ | 160 | L | Insert at the start of a sequence |
| . | | 220 | R | Renaming or data separator used in the operators $\forall$, $\exists$, $\cup$, $\cap$, $\Sigma$, $\Pi$, $\lambda$ |
| .. | | 170 | L | Interval |
| / | | 190 | L | Integer division |
| /: | $\notin$ | 160 | L | Non-belonging |
| /<: | $\not\subseteq$ | 110 | L | Non-inclusion |
| /<<: | $\not\subset$ | 110 | L | Strict non-inclusion |
| /= | $\neq$ | 160 | L | Not equal |
| /\ | $\cap$ | 160 | L | Intersection |
| /\|\ | $\uparrow$ | 160 | L | Restriction of a sequence to the head |
| : | $\in$ | 60 | L | Belonging |
| : | | 120 | L | Record field |
| :: | $:\in$ | | L | Becomes part of (belonging) |
| := | | | L | Becomes equal to |
| ; | | 20 | L | Sequencing for substitution or composition of relations |

| ASCII | Math. | Pri. | As. | Description |
|---|---|---|---|---|
| < | | 160 | L | Strictly lesser than, or definition file delimiter |
| <+ | ⩤ | 160 | L | Overload a relation |
| <-> | $\leftrightarrow$ | 125 | L | Set of relations |
| <- | $\leftarrow$ | 160 | L | Insert at end of sequence |
| <-- | $\longleftarrow$ | | L | Operation output parameters |
| <: | $\subseteq$ | 110 | L | Inclusion |
| <<: | $\subset$ | 110 | L | Strict inclusion |
| <<\| | ⩤ | 160 | L | Subtraction to the domain |
| <= | $\leq$ | 160 | L | Lesser than or equal to |
| <=> | $\Leftrightarrow$ | 60 | L | Equivalence |
| <\| | $\lhd$ | 160 | L | Restriction to the domain |
| = | | 60 | L | Equals |
| == | | | | Definition |
| => | $\Rightarrow$ | 30 | L | Implies |
| > | | 160 | L | Strictly greater than, or definition file delimiter |
| >+> | ⤔ | 125 | L | Partial injection |
| >-> | $\rightarrowtail$ | 125 | L | Total injection |
| >->> | ⤖ | 125 | L | Total bijection |
| >< | $\otimes$ | 160 | L | Direct product of relations |
| >= | $\geq$ | 160 | L | Greater than or equal to |
| ABSTRACT_CONSTANTS | | | | ABSTRACT_CONSTANTS clause |
| ABSTRACT_VARIABLES | | | | ABSTRACT_VARIABLES clause |
| ANY | | | | ANY substitution |
| ASSERT | | | | ASSERT substitution |
| ASSERTIONS | | | | ASSERTIONS clause |
| BE | | | | LET substitution |
| BEGIN | | | | BEGIN substitution |
| BOOL | | | | Set of the Boolean values |
| CASE | | | | CASE substitution |
| CHOICE | | | | CHOICE substitution |
| CONCRETE_CONSTANTS | | | | CONCRETE_CONSTANTS clause |
| CONCRETE_VARIABLES | | | | CONCRETE_VARIABLES clause |
| CONSTANTS | | | | CONSTANTS clause |
| CONSTRAINTS | | | | CONSTRAINTS clause |

| ASCII | Math. | Pri. | As. | Description |
|---|---|---|---|---|
| DEFINITIONS | | | | DEFINITIONS clause |
| DO | | | | WHILE substitution |
| EITHER | | | | CASE substitution |
| ELSE | | | | IF or CASE substitution |
| ELSIF | | | | IF substitution |
| END | | | | Terminator of clauses or of substitutions BEGIN, PRE, ASSERT, CHOICE, IF, SELECT, ANY, LET, VAR, CASE and WHILE |
| EXTENDS | | | | clause EXTENDS |
| FALSE | | | | Literal Boolean constant “false” |
| FIN | $\mathbb{F}$ | | | Set of finite sub-sets |
| FIN1 | $\mathbb{F}_1$ | | | Set of finite non empty sub-sets |
| IF | | | | Substitution IF |
| IMPLEMENTATION | | | | IMPLEMENTATION clause |
| IMPORTS | | | | IMPORTS clause |
| IN | | | | BE or VAR substitution |
| INCLUDES | | | | INCLUDES clause |
| INITIALISATION | | | | INITIALISATION clause |
| INT | | | | Set of implementable relative integers |
| INTEGER | $\mathbb{Z}$ | | | Set of relative integers |
| INTER | $\bigcap$ | | | Quantified intersection |
| INVARIANT | | | | INVARIANT clause or WHILE substitution |
| LET | | | | LET substitution |
| LOCAL_OPERATIONS | | | | LOCAL_OPERATIONS clause |
| MACHINE | | | | MACHINE clause |
| MAXINT | | | | Largest implementable integer |
| MININT | | | | Smallest implementable integer |
| NAT | | | | Set of implementable natural integers |
| NAT1 | $NAT_1$ | | | Set of non-empty implementable natural integers |
| NATURAL | $\mathbb{N}$ | | | Set of natural integers |
| NATURAL1 | $\mathbb{N}_1$ | | | Set of non-empty natural integers |
| OF | | | | CASE substitution |
| OPERATIONS | | | | OPERATIONS clause |

# B Language Keywords and Operators version 1.8.9

| ASCII | Math. | Pri. | As. | Description |
|---|---|---|---|---|
| `OR` | | | | CHOICE or CASE substitution |
| `PI` | $\Pi$ | | | Quantified integer product |
| `POW` | $\mathbb{P}$ | | | Set of sub-sets |
| `POW1` | $\mathbb{P}_1$ | | | Set of non-empty sub-sets |
| `PRE` | | | | Precondition substitution |
| `PROMOTES` | | | | PROMOTES clause |
| `PROPERTIES` | | | | PROPERTIES clause |
| `REFINES` | | | | REFINES clause |
| `REFINEMENT` | | | | REFINEMENT clause |
| `SEES` | | | | SEES clause |
| `SELECT` | | | | Substitution SELECT |
| `SETS` | | | | SETS clause |
| `SIGMA` | $\Sigma$ | | | Quantified product |
| `STRING` | | | | Set of character strings |
| `THEN` | | | | Precondition substitution, ASSERT, IF or CASE |
| `TRUE` | | | | Literal Boolean constant “true” |
| `UNION` | $\bigcup$ | | | Quantified union |
| `USES` | | | | USES clause |
| `VALUES` | | | | VALUES clause |
| `VAR` | | | | VAR substitution |
| `VARIANT` | | | | WHILE substitution |
| `VARIABLES` | | | | VARIABLES clause |
| `WHEN` | | | | SELECT substitution |
| `WHERE` | | | | ANY substitution |
| `WHILE` | | | | WHILE substitution |
| `[` | | | | Image, or start of sequence |
| `[]` | | | | Empty sequence |
| `\/` | $\cup$ | 160 | L | Union |
| `\\|/` | $\downarrow$ | 160 | L | Restrict a sequence to the end |
| `]` | | | | Image, or end of sequence |
| `^` | $\frown$ | 160 | L | Concatenate sequences |
| `arity` | | | | Tree node arity |
| `bin` | | | | Binary tree in extension |
| `bool` | | | | Predicate boolean cast |
| `btree` | | | | Binary trees |
| `card` | | | | Cardinal |
| `ceiling` | | | | Ceiling function |
| `closure(R)` | $R^*$ | | | Reflexive closure of a relation |
| `closure1(R)` | $R^+$ | | | Closure of a relation |
| `conc` | | | | Concatenation of a succession |
| `const` | | | | Tree constructor |
| `dom` | | | | Domain of a function |
| `father` | | | | Father of a tree node |
| `first` | | | | First element in a sequence |
| `floor` | | | | Floor function |
| `fnc` | | | | Transformed into a function |
| `front` | | | | Front of a sequence |
| `id` | | | | Function identity |
| `infix` | | | | Infix formulae of a tree |
| `inter` | | | | General intersection |
| `iseq` | | | | Set of injective sequences |
| `iseq1` | $\text{iseq}_1$ | | | Set of injective non-empty sequences |
| `iterate(R, n)` | $R^n$ | | | Iteration of a relation |
| `last` | | | | Last element in a sequence |
| `left` | | | | Left tree |
| `max` | | | | Maximum in a set of integers |
| `min` | | | | Minimum in a set of integers |
| `mirror` | | | | Mirror of a tree |
| `mod` | | 190 | L | Modulo |
| `not` | $\neg$ | | | Logical not |
| `or` | $\vee$ | 40 | L | Disjunction (logical OR) |
| `perm` | | | | Set of permutations (bijective sequences) |
| `postfix` | | | | Postfix formulae of a tree |
| `pred` | | | | Predecessor of an integer |
| `prefix` | | | | Prefix formulae of a tree |
| `prj1` | $\text{prj}_1$ | | | First projection of a relation |
| `prj2` | $\text{prj}_2$ | | | Second projection of a relation |
| `ran` | | | | Range of a relation |
| `rank` | | | | Rank of a tree node |
| `real` | | | | Conversion from integer to real |
| `rec` | | | | Record in extension |
| `rel` | | | | Relation transform |
| `rev` | | | | Reverse of a sequence |
| `right` | | | | Right tree |
| `seq` | | | | Set of sequences |
| `seq1` | | | | Set of non-empty sequences |
| `size` | | | | Size of a sequence |
| `sizet` | | | | Size of a tree |
| `skip` | | | | Null substitution |
| `son` | | | | ith son of a tree |
| `sons` | | | | Sons of a tree node |
| `struct` | | | | Set of records |
| `subtree` | | | | Subtree of a tree |
| `succ` | | | | Successor |
| `tail` | | | | Tail of a sequence |
| `top` | | | | Top of a tree |
| `tree` | | | | Trees |
| `union` | | | | Generalized union |
| `{` | | | | Start of set |
| `{}` | $\emptyset$ | | | Empty set |
| `\|` | | 10 | L | Vertical bar used in $\forall$, $\exists$, $\cup$, $\cap$, $\Sigma$, $\Pi$, $\lambda$, $\{ \mid \}$ |
| `\|->` | $\mapsto$ | 160 | L | Maplet |
| `\|>` | $\rhd$ | 160 | L | Restriction to the range |
| `\|>>` | $⩥$ | 160 | L | Subtraction to the range |
| `\|\|` | | 20 | L | Simultaneous substitutions, or parallel product of relations |
| `}` | | | | End of set |
| `r~` | $r^{-1}$ | 230 | L | Reverse relation |

# Bibliography

## Books

[Abr96] J. R. Abrial. *The B-book: assigning programs to meanings*. New York, NY, and USA: Cambridge University Press, 1996 (cited on page 11).

## Articles

[Ben11] Marc V. Benveniste. “On Using B in the Design of Secure Micro-controllers: An Experience Report”. In: *Electr. Notes Theor. Comput. Sci.* 280 (2011), pages 3–22 (cited on page 11).

[Lec08] Thierry Lecomte. “Safe and Reliable Metro Platform Screen Doors Control/Command Systems”. In: LNCS 5014 (2008). Edited by Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, pages 430–434 (cited on page 11).

[Lec09] Thierry Lecomte. “Applying a Formal Method in Industry: A 15-Year Trajectory”. In: LNCS 5825 (2009). Edited by Marıa Alpuente, Byron Cook, and Christophe Joubert, pages 26–34 (cited on page 11).

[Lec16] Thierry Lecomte. “Double cœur et preuve formelle pour automatismes SIL4”. In: *8E-Modèles formels/preuves formelles-sûreté du logiciel* (2016) (cited on page 11).

# Index