CLEARSY

Safety Solutions Designer

Atelier B

The B Workbook

Atelier-B 24.04 (Community Edition)

September 18, 2025

Contents

Glossary **5**

Preface **6**
The B Workbook . . . 6
About the B Method . . . 6
Audience . . . 7
Book organization . . . 8
Environment, tools and setup . . . 9
Project overview and participation . . . 9
Acknowledgement . . . 10

1 A simple railroad switch **11**
1.1 Introduction . . . 11
1.2 Natural language description . . . 11
1.3 Modeling the system . . . 13
1.4 Formal specification in B . . . 14
1.4.1 First project, first machine . . . 15
1.4.2 Specification overview . . . 18
1.5 Implementation . . . 22
1.5.1 Creating an implementation . . . 22
1.5.2 Implementation overview . . . 24
1.6 Proving the implementation . . . 25
1.7 Generating and testing the C code . . . 27
1.8 To go further . . . 33

2 Airlock operating system **34**
2.1 Introduction . . . 34
2.2 Natural language description . . . 34
2.3 Formal specification . . . 35
2.3.1 Model a sensor with Atelier B . . . 36
2.3.2 Create the airlock machine . . . 36
2.3.3 Define the operations . . . 38
2.4 Animation with ProB . . . 41
2.5 Implementation . . . 43
2.6 Proving the implementation . . . 46
2.7 Generating and testing the C code . . . 46
2.8 To go further . . . 53

This document is the property of CLEARSY.

3 Integer arithmetic calculator 54
3.1 Introduction . . . 54
3.2 Natural language description . . . 54
3.3 Formal specification . . . 54
3.3.1 Specification overview . . . 57
3.4 Implementation . . . 58
3.5 Generating C code . . . 63
3.6 To go further . . . 66

4 Fuel level 67
4.1 Introduction . . . 67
4.2 Natural language description . . . 67
4.3 Modular architecture . . . 69
4.4 Formal specification . . . 70
4.4.1 Constants . . . 71
4.4.2 Utils . . . 72
4.4.3 Measurements . . . 73
4.4.4 Main fuel machine . . . 73
4.4.5 Entry-point machine . . . 75
4.5 Implementation . . . 77
4.6 Proving the implementation . . . 80
4.7 Generating and testing the C code . . . 81
4.8 To go further . . . 84

5 A simple loop usage example 85
5.1 Introduction . . . 85
5.2 Natural language description . . . 85
5.3 Formal specification . . . 86
5.4 Implementation . . . 87
5.5 Proving the implementation . . . 89
5.6 Generating and testing the C code . . . 89
5.7 To go further . . . 90

6 Filling an array with a given value 91
6.1 Introduction . . . 91
6.2 Natural language description . . . 91
6.3 Arrays . . . 91
6.3.1 Declaring arrays . . . 91
6.3.2 Domain and image . . . 92

This document is the property of CLEARSY.

6.4 Abstract iterator . . . 92
6.4.1 Abstract machine . . . 92
6.4.2 Constants . . . 94
6.4.3 Implementation of the abstract iterator . . . 95
6.5 Formal specification . . . 97
6.6 Implementation . . . 98
6.7 Proving the implementation . . . 102
6.8 Generating and testing the C code . . . 103
6.9 To go further . . . 104

7 Find the maximum of an array 105
7.1 Introduction . . . 105
7.2 Natural language description . . . 105
7.3 Formal specification . . . 105
7.4 Refinements . . . 106
7.5 Implementation . . . 107
7.6 Proving the implementation . . . 111
7.7 Generating and testing the C code . . . 114
7.8 To go further . . . 115

Appendix 116
Installing ProB2-UI . . . 116
Installing gcc and make on Linux, macOS, and Windows . . . 116
Project configuration . . . 118
Substitutions cheat sheet . . . 121

Bibliography 122

This document is the property of CLEARSY.

Glossary

B0 language: a specific subset of the B language that is designed for translation into compilable code (for example, C). An implementation must be written exclusively in B0. Think of it as similar to using C types, but with no dynamic memory allocation, just statically defined structures, arrays and variables.

ProB: an open-source software tool designed for the formal analysis of system specifications. It serves as an animator, model checker, and constraint solver primarily for the B-Method and Event-B formal languages.

ProB2-UI is a modern, JavaFX-based graphical user interface developed for the ProB tool set, which serves as a model checker, animator, and constraint solver for formal specification languages such as B. It provides an intuitive environment for developing, verifying, and analyzing formal models. Additionally, it checks whether a finite-state model of a system satisfies a given specification-a property known as *correctness*.

This document is the property of CLEARSY.

Preface

The B Workbook

This workbook's aim is to introduce you to the B Method and the Atelier B tool-a powerful development environment based on the B Method. It will display some of the best practices and guide you through the generation of proven C code.

Since the ever-growing use of computer science in the 20th century, software has been used in all domains (finance, transport, leisure, healthcare, etc.). As software began to be used in critical domains-where a malfunction could lead to loss of human life or significant financial damage-it became essential to ensure the reliability of code. At this point, software testing emerged as a key part of development. However, no matter how intensive testing is, it cannot guarantee the absence of bugs.

To address this limitation, formal methods were introduced. In this context, we will explore one such method: the B Method.

About the B Method

B is a formal specification method that employs a dedicated language (detailed in this book) to precisely describe the properties of a given specification, which are often initially written in natural language. Once formalized, these expressions can then be proved to be unambiguous, coherent, and non-contradictory. Subsequently, the user can mathematically prove that these properties are preserved throughout the system's full development.

Therefore, this method and its associated proofs allow for:

1. Achieving clear technical and system specifications structured, coherent and unambiguous.
2. Developing software, contractually guaranteed to be fault-free, in fields such as real-time systems, industrial automation, communication protocols, cryptographic protocols and embedded IT.

The B Method usually refers to the set that includes: The B language, model refinement-which is the process of translating an abstract model into a sequential implementation-their proof and (some of) the related tools.

This document is the property of CLEARSY.

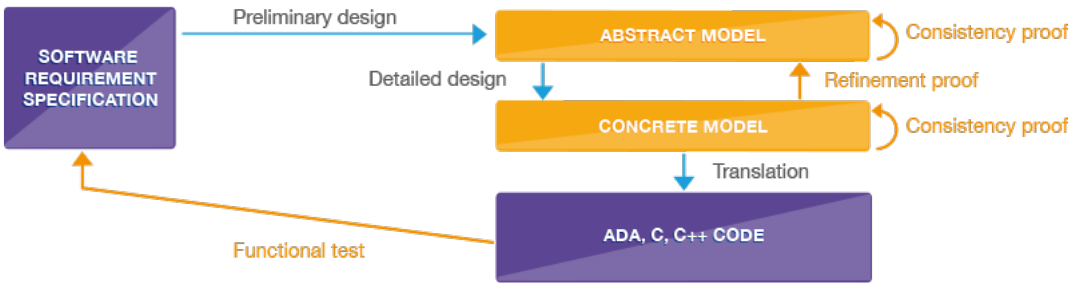


Figure 1: Showcase of a project development with the B Method

B development starts by writing an abstract model including all defined requirements. The model contains the main data processed by the described system. Operations ensure the processing of this data while preserving its properties. The B model thus obtained is called a specification.

The B model is then transformed (refined, in B vocabulary) until a complete software implementation is obtained. Refinement can be done in an iterative manner, allowing atomic changes and lighter proof steps. Indeed, every refinement will require a refinement proof, ensuring that the model properties are conserved troughout the process

Finally, we arrive at a concrete, proven fault-free model that can be coded in C or Ada.

The B Method is therefore: "a proven construction approach (referred to as correct) based on the B language, refinement and proof."

Audience

This workbook is mainly intended for university students in undergraduate and postgraduate formal methods courses, but it can also be useful for anyone studying formal methods, as it covers many key topics in the field. It introduces the Atelier B tool, which is used to design and verify computer and electronic systems using the B Method.

We strongly recommend that you have a basic understanding of logic and set theory before using this workbook. If you're not familiar with these topics, you may want to study some introductory resources[1] to build a solid foundation. It is also beneficial to follow a course in the B Method to get the most out of this workbook[2] [3].

[1] H. B. Enderton, *A Mathematical Introduction to Logic*, 2nd ed. (San Diego: Academic Press, 2001).
[2] Oliveira Marcel and Thierry Lecomte, "The b-Method [MOOC]," n.d., https://mooc.imd.ufrn.br/course/the-b-method.
[3] Steve Schneider, *The b-Method: An Introduction*, Cornerstones of Computing (Palgrave Macmillan, 2001).

This document is the property of CLEARSY.

By using this workbook, you will learn how to apply the B Method and use Atelier B tool in software development and system design. It includes practical exercises and teaches important skills such as formal modeling and verification. This workbook also encourages problem-solving and critical thinking, helping you tackle complex programming tasks.

Book organization

This workbook provides a series of hands-on exercises that aims to guide you step-by-step through learning the B Method and using the Atelier B tool. Each exercise begins with an introduction that outlines the learning objectives and key concepts, followed by a natural language description of the system to be modeled. It then guides you step by step through developing the C implementation and mathematically proving that your software meets its specification. The exercises are designed to be completed in order, with each one building on the concepts introduced earlier.

Each exercise contains a natural language description of a problem that you need to transform into a formal specification using the B Method. This involves defining variables, sets, operations, and properties that capture the system's essential behavior. We advise to create a separate project for each new exercise, and the following workflow : for any component (be it a machine, a refinement, or an implementation) frequently typecheck it, then generate the POs and discharge them in order to avoid mistakes. At this stage, ProB can be used to animate and explore the model, providing valuable feedback for validating its logic.

The specification is then refined step by step, progressing toward an implementation model suitable for translation into C code, with each refinement preserving the original properties. Throughout the process, proof obligations are generated and discharged-either automatically or manually-to establish correctness. Once the implementation is fully proven correct, the resulting code can be generated, compiled, and executed. Optional open-ended questions at the end of each exercise invite further exploration and encourage a deeper understanding of Atelier B and the B Method.

The initial exercises introduce you to the Atelier B interface and the foundational concepts of the B Method, including abstract machines, sets, and operations. You will learn how to create a project, define a machine, and write operations using the B language. Step by step, the exercises guide you through specifying a simple system, verifying its internal consistency, and generating proof obligations. You will also discover how to prove the coherence of your model and ultimately generate C code from an implementation model.

As you work through this workbook, you'll take on more complex systems, learning to model advanced data types, use relations and functions, and animate your models with ProB. You'll see how to break down large problems into smaller parts, organize your projects, and combine modules to build complete, maintainable systems.

This document is the property of CLEARSY.

The exercises gradually introduce you to more advanced concepts, such as loops, invariants, and variants, deepening your understanding of proof and how to handle non-determinism. You will also learn how to perform manual proofs when necessary.

Environment, tools and setup

To complete the exercises in this workbook, you need to install the following tools beforehand:

- Atelier B Community Edition: This is the primary tool used for creating models, performing refinements, and implementing solutions. Download and install the latest version of the Community Edition for your platform by visiting the Atelier B Download Page. Follow the detailed steps in the Installation Guide[4] to complete the setup.
- gcc and Make Environment: These are required for running the C code generated by Atelier B. Ensure that both are installed and configured properly to proceed.
- ProB: ProB is not required for the initial exercises, but it will be needed for later sections. It is recommended to install it at this stage.

Project overview and participation

This project has been initiated by CLEARSY, developer and owner of Atelier B, and industrial practitioner of the B Method. This workbook and its exercises have been tested on ATELIER B 24.04.2 Community Edition, the latest version available at the time. All exercises were tested using the Windows version of Atelier B. The generated C code was compiled and executed in a Unix environment, specifically on Ubuntu running under Windows Subsystem for Linux (WSL).

You can find all the resources for this workbook, including models and code snippets for the exercises, on its GitHub repository. This page provides easy access to everything you need to follow along and get the most out of the workbook. It contains the source code for all the examples and exercises presented in the book, and answers for the exercises are provided.

The workbook will be updated regularly to keep it up to date with new developments in the B Method and the Atelier B tool. These updates might include extra exercises, clearer explanations, and the latest features from Atelier B. We hope user feedback will play an important role in shaping these updates ensuring they meet the needs of learners and improve the overall experience, thus we invite you to contribute to the workbook's development. You can help by suggesting new exercises, refining the content, or proposing topics to add. If you'd like to share your thoughts or ideas, you can use the

[4] CLEARSY, *Installation Guide Community Edition* (CLEARSY, 2024), https://www.atelierb.eu/wp-content/uploads/2024/11/installation-guide-community.pdf.

This document is the property of CLEARSY.

provided email address or submission forms available on the GitHub repository. Your contributions are highly valued and will help make this workbook even better for everyone.

Acknowledgement

We would like to acknowledge the collective effort behind the development and dissemination of this workbook. Its creation was made possible by the collaboration of educators, researchers, and practitioners dedicated to advancing formal methods and safety-critical system engineering.

We would like to thank the people who contributed to sharing knowledge, reviewing content, and testing exercises, and in particular: Guillaume Aichhorn, Dalay Almeida, Lilian Burdy, David Déharbe, Gaëtan Dupeuble, Joao Gouvea-Versiani, Florian Jamain, Thierry Lecomte, Thomas Menier, Marcel Oliveira.

This document is the property of CLEARSY.

1 A simple railroad switch

1.1 Introduction

This first exercise is designed to familiarize you with the **Atelier B** interface and guide you through the structured process of building **proven software** using formal methods.

By working through this exercise, you will:

- Learn how to **navigate and use the Atelier B interface**, becoming familiar with its key components and workflow.
- Specify a basic **stateless function**. A stateless function is a function that does not rely on stored data or previous states. It is a simple yet widely used concept in software modeling.
- **Transform a formal specification into an implementation** using the B language, focusing on writing clauses and defining operations.
- Get an initial understanding of how **type checking** and **proof obligations** function within Atelier B, and how they help verify the correctness of your model.
- Use the **automated proof engine** to prove consistency, learning how to address and resolve proof obligations when needed.
- Learn to **generate C code** directly from your proven model using Atelier B's code generator.
- **Test the generated code** to ensure it behaves according to the specification.

By the end of this exercise, you will have gained practical experience in specifying, implementing, formally proving, and testing software. This foundational work prepares you to take on more advanced challenges in developing complex, verified systems using Atelier B.

1.2 Natural language description

*A **railroad switch** is a mechanical device that enables trains moving from one track to another at a railway junction. The position of the switch is **critical for safety** because an incorrect or ambiguous position can lead to train derailments or collisions.*

In this exercise, we consider a **control system** responsible for reliably **determining the switch's position**. To account for possible sensor failures, the switch is equipped with **three redundant sensors**, each providing a measurement: **m1**, **m2**, and **m3**.

Each sensor can report one of three possible states:

- **Normal**: the switch directs the train onto the **main track**,

This document is the property of CLEARSY.

- **Reverse**: the switch directs the train onto the **diverging track**,
- **Void**: the switch transitions between positions.

The system must determine the switch's position based on the measurements from these sensors. The position is defined as follows:

- The result must be **normal** if at least one measurement is **normal** and none are **reverse**.
- The result must be **reverse** if at least one measurement is **reverse** and none are **normal**.

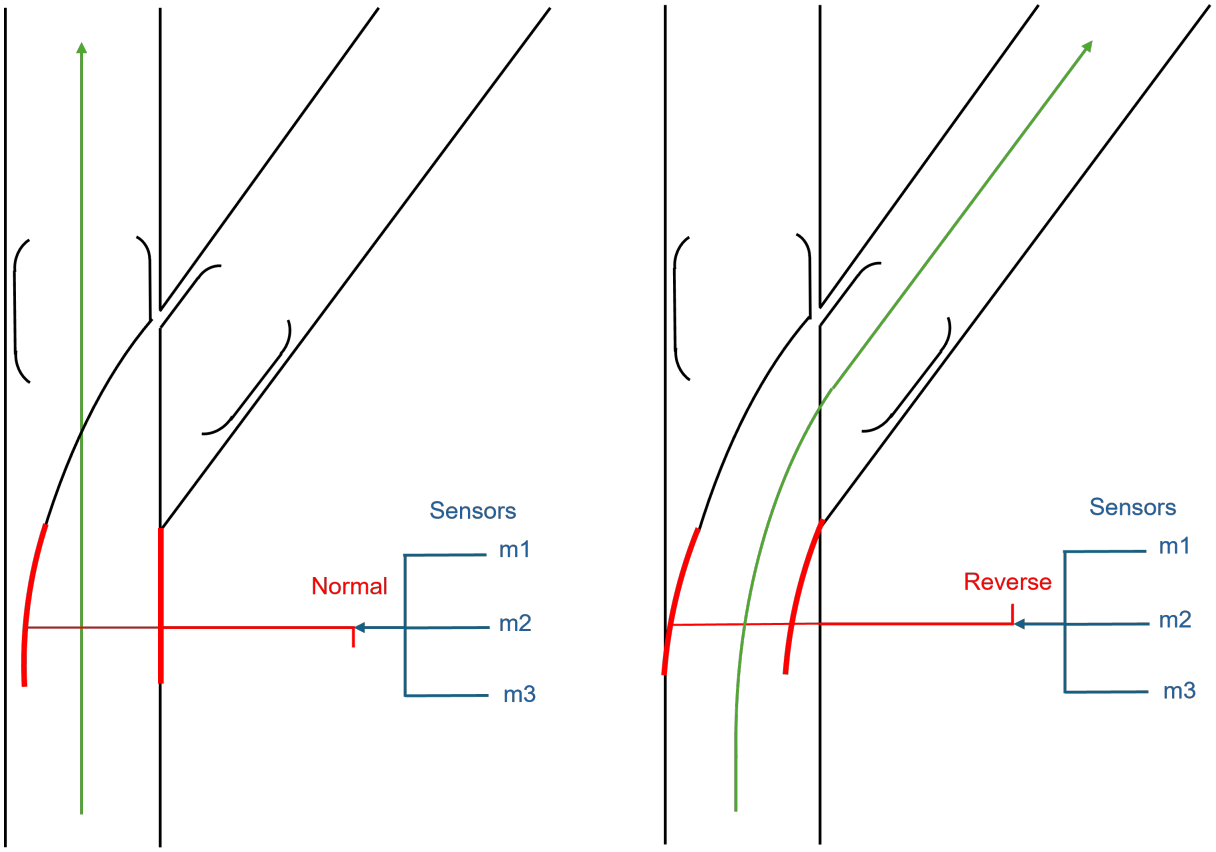


Figure 2: Illustration of a graph representing a railroad switch in normal and reverse position with 3 sensors.

This diagram shows a railroad switch.

- **Green line:** Direction of train travel.
- **Black lines:** Fixed tracks.
- **Red lines:** Moving parts of the switch.

The straight track is called **normal**. The curved track is called **reverse**. Three sensors are used to detect the position of the switch lever.

This document is the property of CLEARSY.

Using three sensors makes it possible to detect simple failures, such as when one sensor provides an incorrect reading. In this situation, the other two sensors can still give reliable information, enabling the system to identify the faulty sensor and determine the correct position. However, if more than one sensor fails at the same time, it becomes much harder to detect errors, and we cannot determine which readings are accurate.

1.3 Modeling the system

The first step in constructing a correct program is to **model** the system. Modeling consists of translating a natural language description into a rigorous mathematical specification, typically expressed as sets, functions, and logical predicates.

Usually, the modeling solution is not unique. Here is one possible way to describe the model :

1. We define an **enumerated set** named `POSITION`, which represents, all possible states for the sensor :

   $$POSITION = \{normal, reverse, void\}$$

2. We specify a **function** named `estimate` that takes three measurements as input and returns an estimated position.

   $$estimate : POSITION \times POSITION \times POSITION \rightarrow POSITION$$

3. We define this function using a **set comprehension**, which is a mathematical notation that allows us to define a set by specifying its elements and the conditions they must satisfy :

   $$\begin{array}{l}\{\,(m_1, m_2, m_3) \mapsto pos \mid \\ m_1, m_2, m_3, pos \in POSITION \times POSITION \times POSITION \times POSITION \wedge \\ (pos = normal \implies normal \in \{m_1, m_2, m_3\} \wedge reverse \notin \{m_1, m_2, m_3\}) \wedge \\ (pos = reverse \implies reverse \in \{m_1, m_2, m_3\} \wedge normal \notin \{m_1, m_2, m_3\}) \\ \}\end{array}$$

   - `m1`, `m2`, and `m3` are the three measurements from the sensors, and `pos` is the estimated position of the switch.
   - The first condition states that if at least one of the measurements is `normal` and none are `reverse`, then the position is `normal`.
   - The second condition states that if at least one of the measurements is `reverse` and none are `normal`, then the position is `reverse`.

This document is the property of CLEARSY.

The B philosophy encourages you to take advantage of high-level mathematical structures to make your model both simpler and clearer to read and review. A good specification should closely follow the original system description. Because the coherence between the natural language description and the specification can only be done by manual proofreading.
While later, refinements and implementations may differ in structure or detail, what is important is that you can formally prove each one remains consistent with the original specification.

1.4 Formal specification in B

The next step is to express the mathematical model in the B language. The B Method uses a **refinement-based approach** to develop software.

First, you begin with an **abstract machine**-a high-level model that captures the essential behavior of your system.This model is then refined step by step, transforming it gradually while ensuring that each transformation preserves the original behavior. This coherence is maintained by proving the correctness of each refinement.

The model can have zero or more **refinements**. They are used to gradually transform the specification into code by adding more and more implementation details. Refining in several steps helps to simplify the proof and makes the transition from specification to code clearer. In B, there are generally very few refinements, because the goal is to decompose the software into simple modules that are therefore relatively close to their implementation.

The last refinement is called an **implementation**. It is the last step before generating the code. The implementation is written in a specific subset of the B language, called B0, which is designed to be easily translatable into running code, such as C.

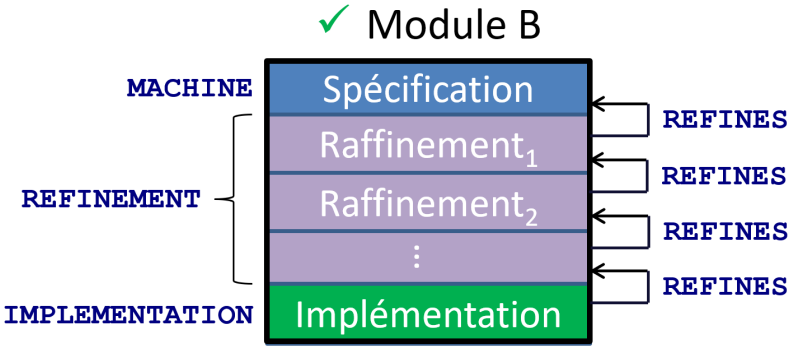


Figure 3: Structure of a module in B.

This document is the property of CLEARSY.

Correctness is ensured through a series of **proof obligations**. These are mathematical propositions that must be proven to show that the refinement maintains the properties of the original specification. Each refinement step generates proof obligations that ensure the new, more detailed model remains consistent with the higher-level specification it refines. By proving all these obligations, the B Method guarantees that the final implementation correctly satisfies the original specification.

1.4.1 First project, first machine

Click on Atelier B app to run it, you should see a window as seen below.

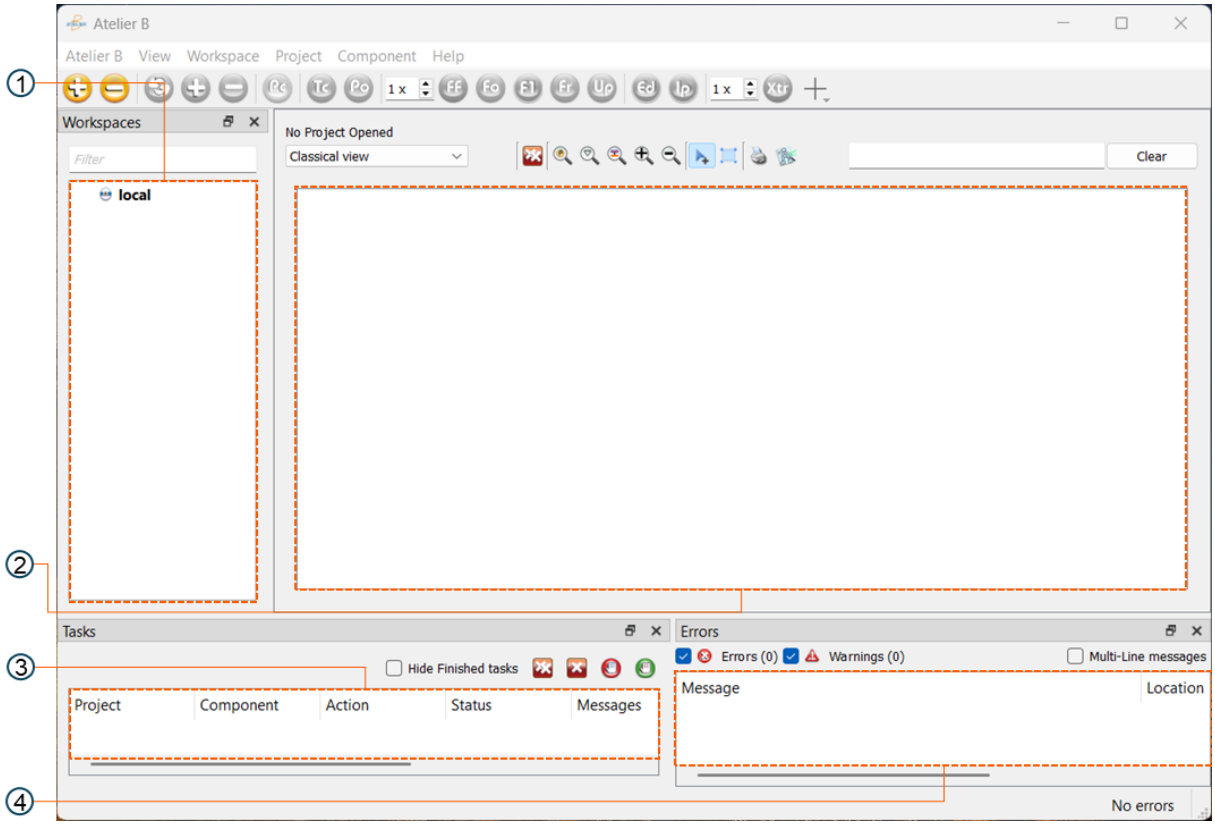


Figure 4: Main Window of Atelier B

Atelier B is project-based:

1. Although empty for the moment, this part shows all created projects;
2. In this panel, all opened project components are presented, as well as their status;
3. Here, you may find lists of tasks executed by Atelier B;
4. The last panel shows all different error messages.

This document is the property of CLEARSY.

Create a new project

1. To create a new project, go to the menu **Atelier B** > **New** > **Project...**.
2. Give a name for your project, for example "switch", and select "Software development".
3. If this is your first time, the next window will ask you to specify the workspace directory. This is where new components and internal files required by Atelier B will be stored. Make sure to choose a directory where you have full read and write permissions (for example, a folder inside your home directory). If you select a location where you lack sufficient permissions (such as a system or administrator-protected folder), Atelier B may not function correctly and could display errors. You can keep the default value or change it as needed, but always ensure you have the necessary access rights.
4. At last, a final menu allows you to configure some global options for the project. Once again you can keep all default settings and click on "Finish".

Important: Before proceeding, make sure to **follow the additionnal instructions in the appendix** under Project configuration. These instructions are **mandatory** to follow the exercises in this workbook. They help display the Atelier B interface correctly and ensure Atelier B run without issues.

Create a new machine

1. To create a new component, go to the menu **Atelier B** > **New** > **Component...**.
2. Enter a name for your component (for example, "switch") and select "Machine" as the component type. You can keep the default directory.
3. The last panel gives you a preview of your new component. Click "Finish" to create it.
4. The created component will appear in the main window. We recommend using the "Classical View" rather than the graphical vertical view.
5. Double-click the component on the main view to open it in the integrated Atelier B component editor.

This document is the property of CLEARSY.

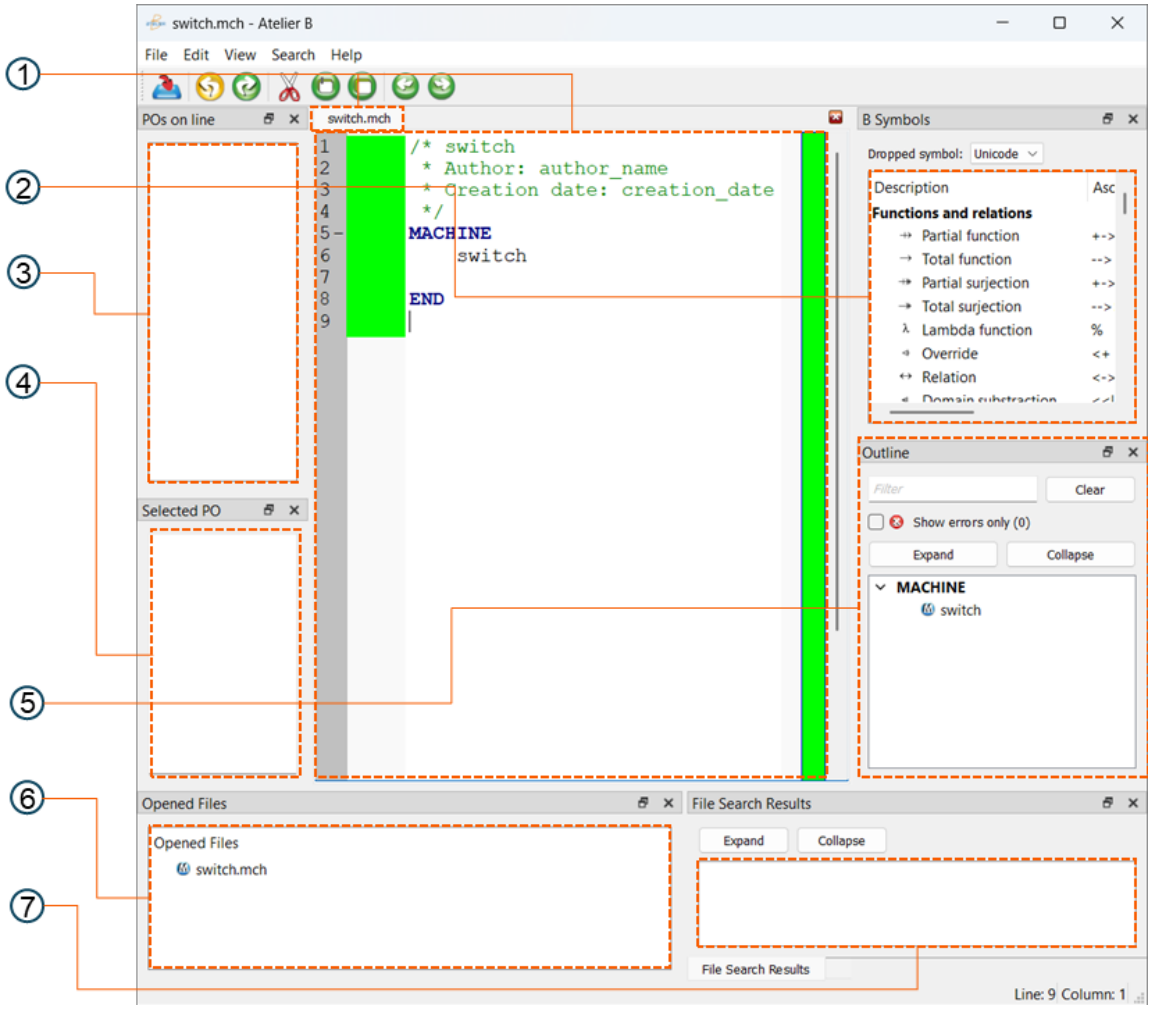


Figure 5: Machine editor view

A new window will open, divided into several functional panels:

1. The main area is a text editor where you edit your machines. You can switch between open machines by clicking on their names above the editor.
2. Next to the text editor, you will find a comprehensive list of B symbols in both Unicode and ASCII. Use this resource to help you write your machines.
3. Displays a list of generated proof obligations, including the specific lines related to each.
4. Displays the details of the current selected proof obligation that needs to be proven.
5. Shows any errors detected by Atelier B. If problems occur, check this panel to identify and troubleshoot issues.
6. List of all open files.

This document is the property of CLEARSY.

7. Provides a file search function, making it easy to locate specific files.

A machine consists of different **clauses**. You start it with a `MACHINE` clause, follow with the name of the component, and end with a `END` clause.

By itself, this machine does not specify anything. Copy the formal specification below into the machine editor. This specification directly reflects the mathematical model described earlier.

```
MACHINE switch
SETS
    POSITION = {normal,reverse,void}
OPERATIONS
pos <-- estimate(m1,m2,m3) =
    PRE
        m1: POSITION &
        m2: POSITION &
        m3: POSITION
    THEN
        pos:(
            pos: POSITION &
            (pos = normal => (normal: {m1,m2,m3} & reverse /: {m1,m2,m3})) &
            (pos = reverse => (reverse: {m1,m2,m3} & normal /: {m1,m2,m3}))
        )
    END
END
```

1.4.2 Specification overview

The SETS Clause

To define a set, add a `SETS` clause to the machine. This clause allows you to declare an abstract set or to list its elements explicitly (an enumerated set).

```
SETS
    POSITION = {normal, reverse, void}
```

This line introduces an enumerated set called `POSITION`, which includes the three distinct states the switch may occupy: `normal`, `reverse`, and `void`.

For more information on the SETS clause and syntax, refer to the B Language Reference Manual[5], page 136.

[5] CLEARSY, *B Language Reference Manual*, Version 1.8.10 (Aix-en-Provence, France, n.d.), https://www.atelierb.eu/wp-content/uploads/2023/10/b-language-reference-manual.pdf.

This document is the property of CLEARSY.

The OPERATIONS Clause

In a specification, the `OPERATIONS` clause defines the behavior of one or more operations, which may update components, modify variables, or assign values to output parameters.

Each operation follows a specific structure:

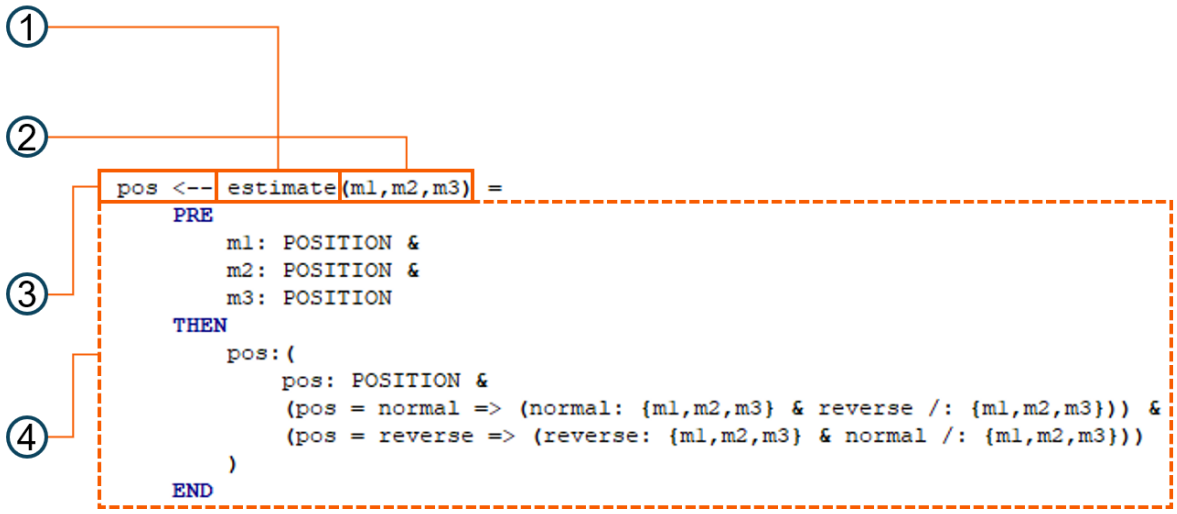


Figure 6: B operation structure

1. The operation name.
2. Input parameters, if any. Unlike many programming languages, in B, you do not write parentheses when there are no input parameters.
3. The output parameter, which is optional.
4. A **substitution**, which is a mathematical expression that defines the operation's behavior.

A machine can contain multiple operations, separated by ;. Unlike many programming languages such as C, ; is just a separator-not an instruction terminator. Do not place ; at the end of the last operation.

Substitutions in B

Substitutions are mathematical expressions that describes how the state of the system changes when an operation is executed. It can be thought of as a set of rules that define how input values are transformed into output values.

The Precondition Substitution

One substitution you will often use is the **precondition substitution**. Its syntax:

This document is the property of CLEARSY.

`PRE` *Predicate* `THEN` *Substitution* `END`

Precondition substitution is used to set the conditions required to call an operation. If the *predicate* of the precondition is false, executing the substitution leads to undefined behavior. This is useful for ensuring that the operation is only called when certain conditions are met. In other words, the behavior described by a substitution with a precondition is guaranteed only if the predicate is true whenever the operation is called.

Predicates

A **predicate** is a logical formula that expresses a property or condition about data. In the B language, predicates are fundamental for specifying and constraining system behavior. They are used to:

- **Define properties of data**: Predicates appear in clauses such as `CONSTRAINTS`, `PROPERTIES`, `INVARIANT`, and `ASSERTIONS` to formally state requirements, invariants between variables.
- **Express conditions when applying substitutions** (`PRE`, `SELECT`, `IF`, `WHILE` substitutions).

Predicates can include quantifiers such as $\forall$ (for all) and $\exists$ (there exists), as well as constructs like λ-expressions, set comprehensions (`{|}`), summations (Σ), products (Π), unions ($\cup$), and intersections ($\cap$).

"Becomes such that" Substitution

The "becomes such that" substitution is used to replace variables with values that satisfy a given predicate. The variables must be pairwise distinct. If several values satisfy the predicate, the substitution does not specify which one will be chosen; its behavior will then be non-deterministic :

Let P be a predicate, X a list of modifiable variables that are pairwise distinct, Y a list of intermediate variables with as many elements as X but that are not in X and P, then:

```
X :(P) = ANY Y WHERE [X := Y]P THEN X := Y END
```

In the expression X :(P), the variable list X must be typed in the predicate P with the help of abstract data typing predicates located in a conjunction list at the highest level of the syntax analysis of P.

```
pos:(
    pos: POSITION &
    (pos = normal => (normal: {m1,m2,m3} & reverse /: {m1,m2,m3})) &
    (pos = reverse => (reverse: {m1,m2,m3} & normal /: {m1,m2,m3}))
)
```

The first line of the predicate, `pos: POSITION`, specifies the type of `pos`. The following lines define the logical conditions and constraints that determine the correct value of `pos` based on the sensor measurements. Each condition corresponds directly to a rule from the natural language specification.

Many more substitutions can be used within a machine. Upcoming exercises will introduce additional examples. For a comprehensive list and detailed explanations, refer to the B Language Reference Manual[6].

Coherence proof

The B Method uses mathematics to specify software components. Furthermore, it confirms that this description is correct by requiring you to prove that the propositions are valid. Here is how you must proceed each time:

- First, we need to type-check the components. This will confirm that both the B syntax and grammar are correct, that all identifiers are known, and verify that all variables are used where they must be used according to their types.

Type-checking is done on selected components by clicking on the  button on the main view.

Once done, you should see an OK in the "Typechecked" column. Our goal is to validate all the steps all the way to "B0 Checked".
Atelier B automatically refreshes the view; you can force it by using F5 once done.

- The next step consists of generating proof obligations. To prove the coherence of our model, Atelier B generates mathematical propositions, called proof obligations, that we need to prove. If they are all proved, then according to the B Method, the model is coherent.

Proof obligations can be generated by clicking on the  button once the component is selected.

Finally, you must prove the proof obligations. You can prove a machine automatically using the  button for a simple prover, or the  button for a more advanced one-although the advanced prover might take longer. When you click either  or  , Atelier B will automatically type-check, generate proof obligations, and run the selected prover. Similarly, saving the file triggers Atelier B to attempt a proof using the 'f0 prover', if your Atelier B is properly configured according to the appendix.

[6] CLEARSY, *B Language Reference Manual*, https://www.atelierb.eu/wp-content/uploads/2023/10/b-language-reference-manual.pdf.

This document is the property of CLEARSY.

Automatically proving all valid proof obligations of a model is a very complex mathematical problem. Currently, we are only able to prove some of these automatically. For the remaining proof obligations, you will learn how to handle them manually. Keep in mind that unproven proof obligations are either invalid or cannot be demonstrated automatically by the proof tool. In such cases, you need to review the proof obligations to determine whether they are valid. If they are invalid, you need to correct your model. If they are valid, you have the choice to either reformulate your model (to ease automatic proof) or to enter interactive proof.

Since the machine only contains a constant enumerated set and a stateless function, there isn't additional model consistency to verify. Hence, no proof obligations are generated. And since there are no proof obligations generated, there's nothing to prove.

After saving the machine with `Ctrl+S`, the main view should display "Typechecked" and "POs Generated" with "OK" status next to the `switch` component. You will also see that there are 0 "Proof Obligations" and therefore 0 "Proved", and 0 "Unproved".

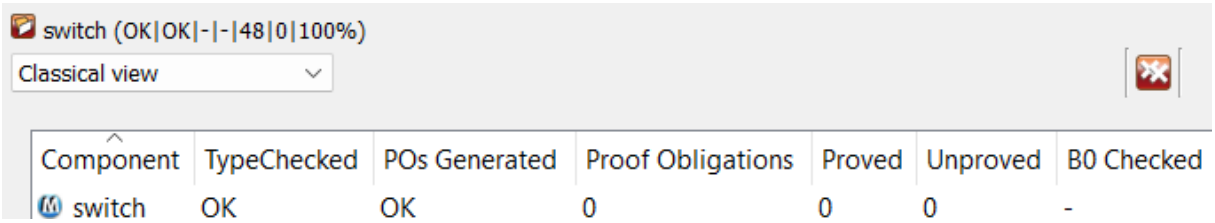


Figure 7: Machine proof obligations

1.5 Implementation

We've reached the final stage of the B Method before automatically translated our B code into a desired language: transforming abstract specifications into something that directly resembles a programming language. In this first and final refinement-what we call an implementation-you must express the data and operations using constructs that programming languages can understand, rather than the abstract mathematical notions seen in previous stages. To do this, we use B0 language.

1.5.1 Creating an implementation

1. Go to the menu **Atelier B** > **New** > **Component...**.
2. Select "implementation" for the type of machine.
3. Ensure the machine selected in the "refines" section is the previously created machine.
4. By default, the name for an implementation is the name of the machine with "_i". It should be automatically filled.

This document is the property of CLEARSY.

In Atelier B, creating a machine generates a .mch file, while creating an implementation generates a .imp file. This naming and file extension convention helps distinguish between the abstract specification and its implementation.

5. In the next panel, you'll see options to copy parameters from the abstract machine. In this project, you're not interested in copying any parameters, so leave it as is and click Next.
6. The final panel shows a preview of your new implementation. Click Finish to create it.
7. Double-click the new component in the main view to open it in the integrated component editor in Atelier B.

Copy the code below into the new component.

```
IMPLEMENTATION switch_i
REFINES switch
OPERATIONS
    pos <-- estimate(m1,m2,m3) =
    CASE m1 OF
        EITHER normal THEN
            IF m2 = reverse or m3 = reverse THEN
                pos:=void
            ELSE
                pos:=normal
            END
        OR reverse THEN
            IF m2 = normal or m3 = normal THEN
                pos:=void
            ELSE
                pos:=reverse
            END
        ELSE
            CASE m2 OF
                EITHER normal THEN
                    IF m3 = reverse THEN
                        pos:=void
                    ELSE
                        pos:=normal
                    END
                OR reverse THEN
                    IF m3 = normal THEN
                        pos:=void
                    ELSE
                        pos:=reverse
```


This document is the property of CLEARSY.

```
                    END
                ELSE
                     CASE m3 OF
                        EITHER normal THEN
                            pos:=normal
                        OR reverse THEN
                            pos:=reverse
                        OR void THEN
                            pos:=void
                        END
                    END
                END
            END
        END
    END
END
```

1.5.2 Implementation overview

Assigning Values to Constants and Sets

We normally need to assign values to our constants and abstract sets in a `VALUES` clause. The goal is to provide explicit assignments to constants and abstract sets, and to avoid specifying miracles i.e., constants that cannot be instantiated, such as a Boolean constant that is simultaneously equal to both TRUE and FALSE.

However, there is one exception: enumerated sets. These sets don't need explicit values because their declaration already serves as their valuation. When defined, they are automatically translated into enumerated types in the implementation.

Operations

The 'becomes such that' substitution does not belong to the B0 language and therefore cannot be directly translated into executable code. Instead, we use a B0 substitution. A complete list of all substitutions can be found in the appendix section titled 'Substitutions Cheat Sheet', which covers those applicable to specifications, implementations, or both.

In this case, where you need conditional execution, the `CASE OF` structure is used. It allows you to handle multiple conditions efficiently, ensuring that different scenarios are properly covered within your implementation.

This document is the property of CLEARSY.

1.6 Proving the implementation

Select the component then click on the  button to prove automatically.

After all proof obligations (POs) are proved, Atelier B normally highlights the editor with green stripes in the editor bar, with a list of all POs on the side bar. Due to a known display issue in this exercise, the PO list may not appear in the editor even if PO display options are enabled. To confirm that all POs are generated and proved, refer to the main view.

Here is how the editor should appear:

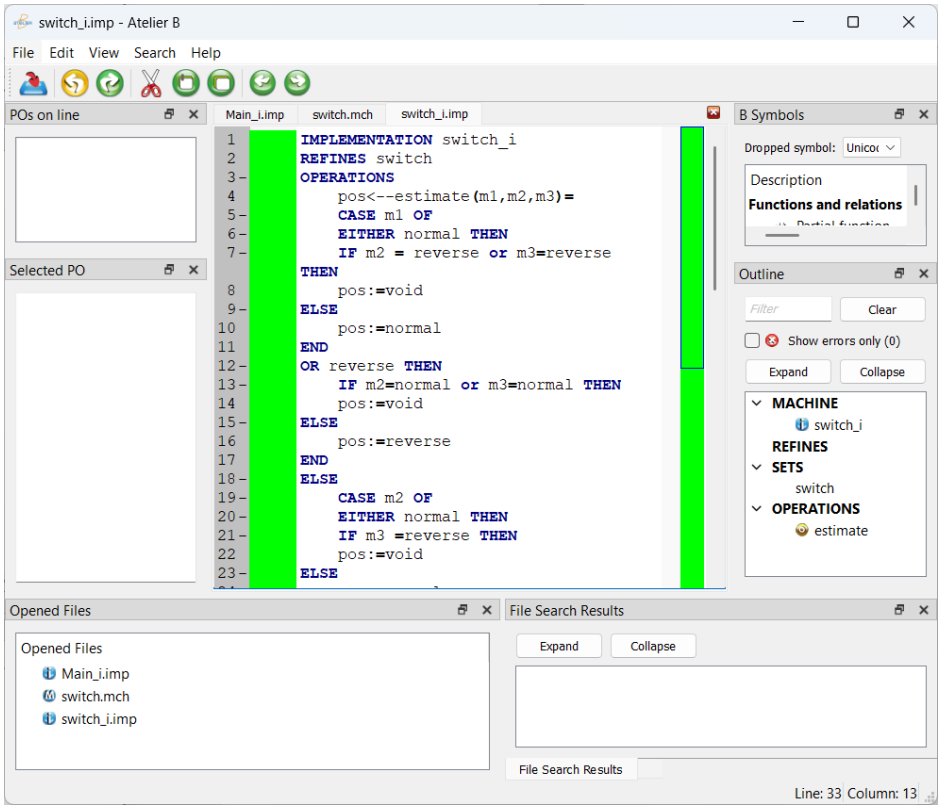


Figure 8: Editor with implementation anomaly

For your reference, the screenshots below illustrate how the editor should in normal conditions, displaying the proof obligations and their details.

This document is the property of CLEARSY.

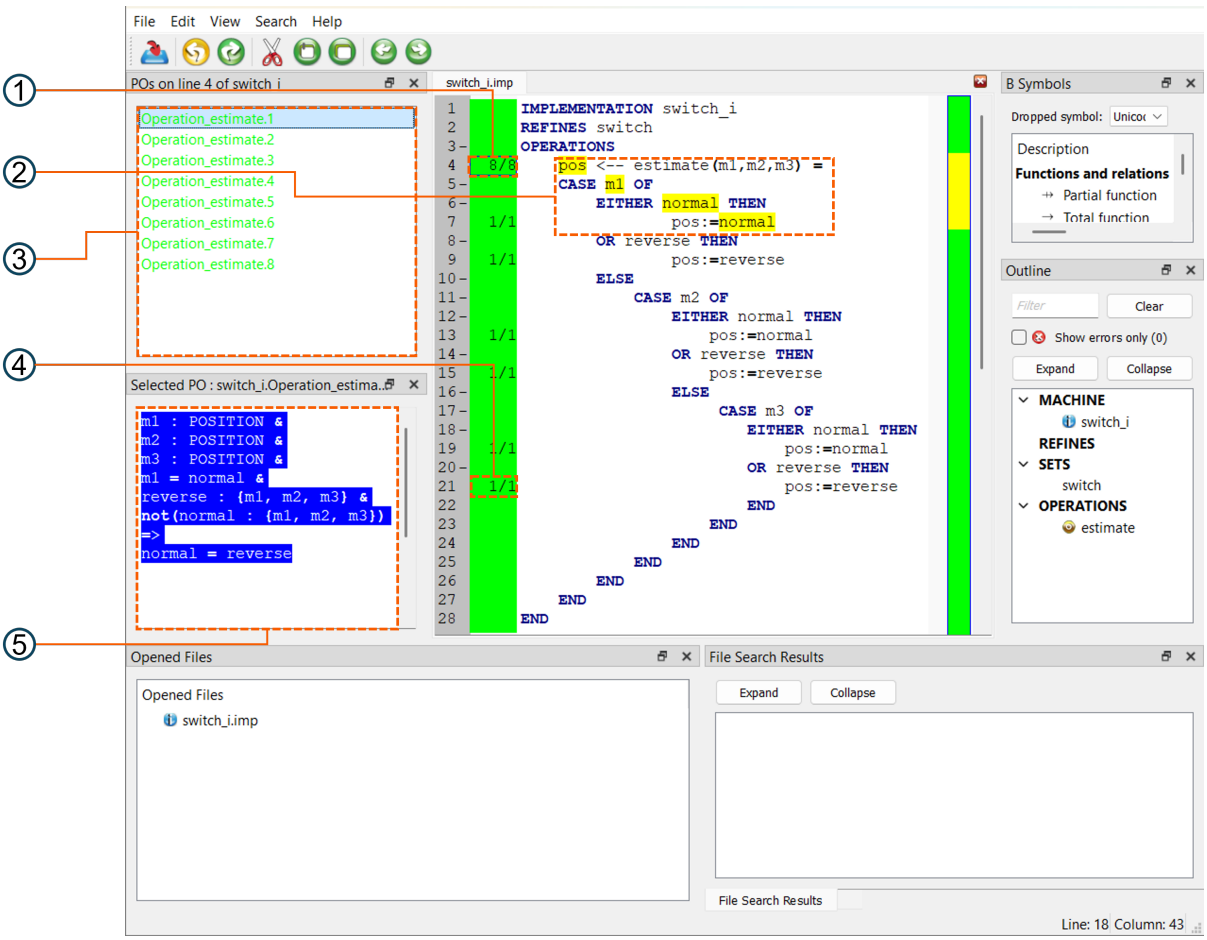
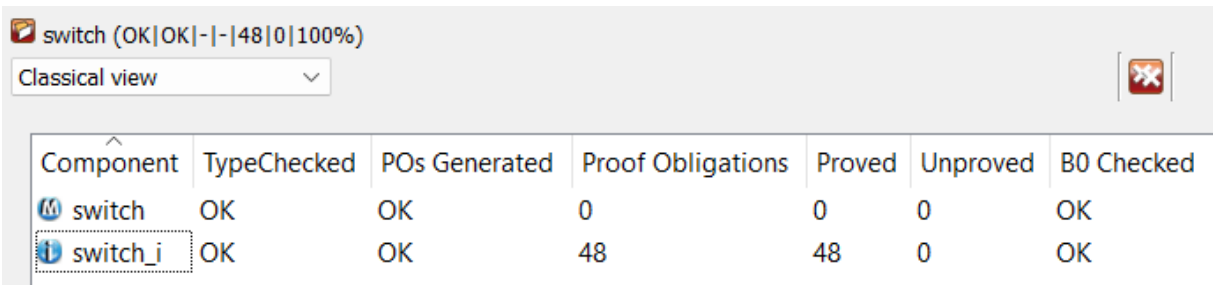


Figure 9: Editor with proof obligations displayed

1. Next to each declaration, you can see the number of proof obligations related to the operation. This shows how many have been proved out of the total.
2. When you select a proof obligation, the related parts of the model are automatically highlighted.
3. Displays a list of all proof obligations.
4. You can also see the number of proof obligations associated with each line.
5. Displays the details of the selected proof obligation.

Get back to the main view. You can see next to the `switch_i` implementation, "TypeChecked" and "POs Generated" with OK status, as well as 48 "Proof Obligations", all proved. You can then select the component and go to **Component** > **B0 Check**. The component is now B0 Checked. Here is how the main view should look like:

This document is the property of CLEARSY.



Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
switch	OK	OK	0	0	0	OK
switch_i	OK	OK	48	48	0	OK

Figure 10: Implementation B0 Check Status

1.7 Generating and testing the C code

Once Atelier B has proved the implementation to be consistent with the specification and confirmed it is using the B0 language, we can generate the associated C code with the built-in translator.

Note that you can still generate code with Atelier B even if the code is not proved; however, this completely invalidates the purpose of using Atelier B and formal methods.

1. Go to the menu **Project** > **Code generator**.
2. Select the "C" translator.
3. Use the "C9X" translation profile (it is a past version of the C programming language open standard released in 1999), then click OK.
4. Wait for the process to finish.
5. When the process completes, go to **Project** > **Open Folder** to access the generated files in the `lang/c` folder.

The resulting C code, generated by the B0 to C translator from the implementation must equals the one given below:

switch.h

```
/* File switch.h */
#ifndef _switch_h
#define _switch_h

#include <stdint.h>
#include <stdbool.h>
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

```
/* Clause SETS */
typedef enum
{
    switch__normal,
    switch__reverse,
    switch__void

} switch__POSITION;
#define switch__POSITION__max 3

/* Clause CONCRETE_CONSTANTS */
/* Basic constants */
/* Array and record constants */

/* Clause CONCRETE_VARIABLES */

extern void switch__INITIALISATION(void);

/* Clause OPERATIONS */

extern void switch__estimate(switch__POSITION m1, switch__POSITION m2,
 ↪  switch__POSITION m3, switch__POSITION *pos);

#ifdef __cplusplus
}
#endif /* __cplusplus */


#endif /* _switch_h */
```

switch_i.c

```
/* File switch_i.c */

#include "switch.h"

/* Clause CONCRETE_CONSTANTS */
/* Basic constants */

/* Array and record constants */
```


This document is the property of CLEARSY.

```
/* Clause CONCRETE_VARIABLES */

/* Clause INITIALISATION */
void switch__INITIALISATION(void)
{

}

/* Clause OPERATIONS */

void switch__estimate(switch__POSITION m1, switch__POSITION m2, switch__POSITION m3,
 ↪  switch__POSITION *pos)
{
    switch(m1)
    {
        case switch__normal : {

            if((m2 == switch__reverse) ||
            (m3 == switch__reverse))
            {
                (*pos) = switch__void;
            }
            else
            {
                (*pos) = switch__normal;
            }
            break;
        }
        case switch__reverse : {

            if((m2 == switch__normal) ||
            (m3 == switch__normal))
            {
                (*pos) = switch__void;
            }
            else
            {
                (*pos) = switch__reverse;
            }
            break;
        }
        default : {
```


This document is the property of CLEARSY.

```
switch(m2)
{
    case switch__normal : {

        if(m3 == switch__reverse)
        {
            (*pos) = switch__void;
        }
        else
        {
            (*pos) = switch__normal;
        }
        break;
    }
    case switch__reverse : {

        if(m3 == switch__normal)
        {
            (*pos) = switch__void;
        }
        else
        {
            (*pos) = switch__reverse;
        }
        break;
    }
    default : {

        switch(m3)
        {
            case switch__normal : {

                (*pos) = switch__normal;
                break;
            }
            case switch__reverse : {

                (*pos) = switch__reverse;
                break;
            }
            case switch__void : {
```


This document is the property of CLEARSY.

```
                        (*pos) = switch__void;
                        break;
                    }
                }
                break;
            }
        }
        break;
    }
}
```

The `CASE OF` clause is now replaced with `switch case` in C.

To test the generated code, place the provided `main.c` file and the Makefile in the same directory as your generated code. These files can be found either in the section below or in the GitHub repository, in the `src/a_simple_switch/generated_code` folder. Alternatively, you can generate the main file directly from your B project. The provided main file tests and displays all possible sensor combinations, along with the results from the `estimate` function.

```c
/* File main.c */
#include <stdio.h>
#include "switch.h"

void printPosition(switch__POSITION pos) {
    switch(pos) {
        case switch__normal:    printf(" normal"); break;   // 0
        case switch__reverse:   printf("reverse"); break;   // 1
        case switch__void:      printf("   void"); break;   // 2
    }
}

int main (void) {

    switch__POSITION m1;
    switch__POSITION m2;
    switch__POSITION m3;
    switch__POSITION pos;
    switch__POSITION *ppos = &pos;
```

```
    printf("+==========+==========+==========++==========+\n");
    printf("|    m1    |    m2    |    m3    ||    pos    |\n");
    printf("+==========+==========+==========++==========+\n");

    for(m1 = 0; m1 < 3; m1++) {
        for(m2 = 0; m2 < 3; m2++) {
            for(m3 = 0; m3 < 3; m3++) {
                switch__estimate(m1, m2, m3, ppos);
                printf("|  ");printPosition(m1);printf(" |
↪  ");printPosition(m2);printf(" |  ");printPosition(m3);printf(" ||
↪  ");printPosition(pos);printf(" |\n");
            }
        }
    }

    printf("+==========+==========+==========++==========+\n");

    return 0;
}
```

MakeFile

```
objects = switch.o main.o
all: $(objects)
    $(CC) $^ -o switch

$(objects): %.o: %.c
    $(CC) -c $^ -o $@

clean:
    rm -f *.o switch
```

You can now compile, run, and test the generated code to ensure it behaves as expected. Once the setup is complete, proceed with the following steps:

- Run `make` to compile the code.
- Then, run `./switch` to execute the main file and display an exhaustive test of the function.
- You can also run `make clean` to remove the output files.

This is the expected execution trace of the program :

This document is the property of CLEARSY.

m1	m2	m3	pos
normal	normal	normal	normal
normal	normal	reverse	normal
normal	normal	void	normal
normal	reverse	normal	normal
normal	reverse	reverse	normal
normal	reverse	void	normal
normal	void	normal	normal
normal	void	reverse	normal
normal	void	void	normal
reverse	normal	normal	reverse
reverse	normal	reverse	reverse
reverse	normal	void	reverse
reverse	reverse	normal	reverse
reverse	reverse	reverse	reverse
reverse	reverse	void	reverse
reverse	void	normal	reverse
reverse	void	reverse	reverse
reverse	void	void	reverse
void	normal	normal	normal
void	normal	reverse	normal
void	normal	void	normal
void	reverse	normal	reverse
void	reverse	reverse	reverse
void	reverse	void	reverse
void	void	normal	normal
void	void	reverse	reverse
void	void	void	reverse

Once you finally reach this step, you are all set to create, prove, and generate your first software with Atelier B.

1.8 To go further

- **Majority voting logic:** Modify the `estimate` function so that it returns the position (normal, reverse, or void) reported by the majority of the three sensors. If all three sensors report different values, the result should be `void`. How would you adapt the B specification and implementation to support this majority-based decision?

This document is the property of CLEARSY.

2 Airlock operating system

2.1 Introduction

By working through this exercise, you will:

- Learn how to **model and use Boolean variables** (`BOOL` type) to represent logical conditions in Atelier B.
- Practice **declaring and typing variables** using the `VARIABLES`, `INVARIANT`, and `INITIALISATION` clauses in a specification machine.
- Understand how to **define and maintain invariants** to ensure system safety properties
- Gain experience in **structuring operations** with preconditions and multiples substitutions to control system behavior.
- Use **ProB** to animate, simulate, and validate the behavior of your B models interactively.
- Implement a **refinement and implementation** of a specification with concrete variable.
- Generate and test **C code** from your proven B implementation.
- **Modeling a sensor using a basic machine:** Learn how to represent a physical sensor in B by defining a *basic machine* and importing it into your machine.

2.2 Natural language description

An airlock is a room or compartment that allows passage between environments with different atmospheric pressures or compositions, while minimizing changes between them. An airlock consists of a sealed chamber with two independent, airtight doors, typically arranged in series:

- An **Inner Door** connects the chamber to the pressurized inner environment.
- An **Outer Door** connects the chamber to the lower-pressure outer environment.

To ensure operational safety, **both doors must never be open simultaneously**. Additionally, the chamber's internal pressure must be properly equalized before either door is allowed to open.

This program models the behavior of an **airlock door authorization system**, which determines when each door can be safely opened based on the internal pressure of the chamber.

The machine operates based on these variables:

- `airlock_pressure`: Represents the current pressure value within the airlock chamber. It can take one of three values: `indoor_pressure`, `outdoor_pressure`, or `other_pressure`.
- `is_indoor_door_openable` and `is_outdoor_door_openable`: Boolean variables indicating whether the inner or outer door is allowed to open. A door is authorized to open only if the chamber pressure matches the corresponding environment (inner or outer).

This document is the property of CLEARSY.

- `cycle`: A control variable used to alternate between two operational phases: `ACQ` (acquisition) and `CTRL` (control). This ensures the system regularly alternates between reading the pressure and evaluating door access conditions.

The airlock authorization logic consists of **two operations**: `actualize_pressure` and `enable_doors_opening`. These operations form a simple two-phase control cycle that alternates between **data acquisition** and **authorization decisions**.

During the acquisition phase, the system reads the current pressure of the airlock chamber. This simulates retrieving data from a pressure sensor, switching to the control phase.

In the control phase, the system decides which door-if any-can be safely opened, based on the pressure value obtained during the previous phase:

- The **inner door** is authorized to open if the chamber pressure matches the **inner environment**.
- The **outer door** is authorized to open if the chamber pressure matches the **outer environment**.
- **Other pressure** refers to any pressure value that does not match either the indoor or outdoor environment.

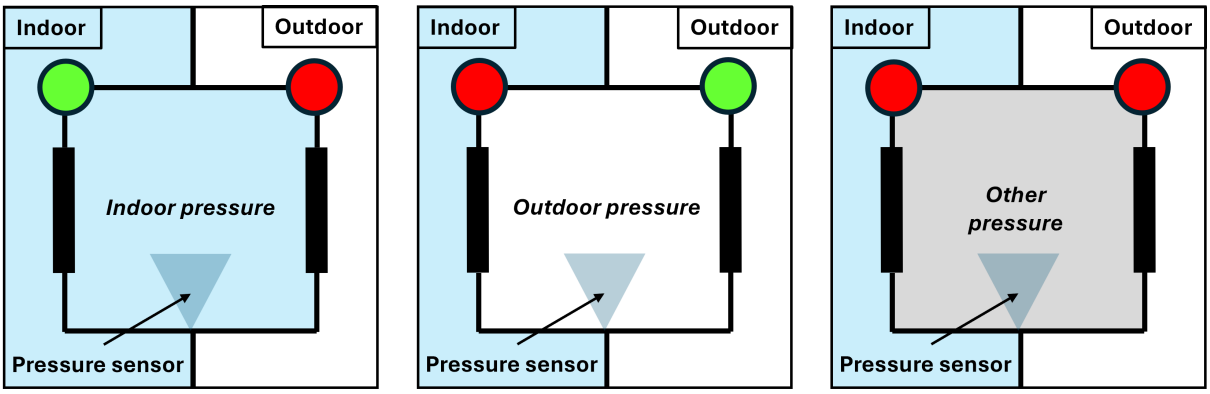


Figure 11: Diagram showing all valid airlock configurations. Green indicates a door is authorized to open; red indicates a door is not authorized.

For example, in the first graph of the top row, the airlock pressure matches the indoor environment. As a result, the system authorize the inner door to be open, while the outer door remains locked.

2.3 Formal specification

First, you shall begin by creating a new project, then create a new machine to model the airlock authorization system which you will complete step-by-step.

This document is the property of CLEARSY.

2.3.1 Model a sensor with Atelier B

To model a sensor in B, you use a **basic machine**. A basic machine is a machine used to **specify the operations of a sensor**, which can be imported and used by other machines.

We use basic machines to model sensors because they provide an abstract, formal interface for sensor operations. Since real-world sensors can be unreliable-subject to noise, faults, or inaccuracies-we cannot formally prove their correctness within the B Method. By modeling sensors as basic machines, we separate the unpredictable physical behavior from the rest of the system, allowing us to reason about and verify the system's logic independently of the actual sensor implementation.

Start by creating a new machine. By convention, the name of a basic machine should end with `_bs`. Name it `Airlock_pressure_bs` to indicate that it models the airlock pressure sensor.

- The airlock pressure is represented by an **enumerated set** called `PRESSURE`, which has three following values `indoor_pressure`,`outdoor_pressure`, `other_pressure`.
- The machine provides a single operation, `get_pressure`, which returns a value from the `PRESSURE` set. This simulates reading the current pressure from the sensor.
- The operation is defined to return any value in `PRESSURE`, reflecting that the sensor could report any of the three possible states.

```
MACHINE
    Airlock_pressure_bs
SETS
    PRESSURE = {indoor_pressure, outdoor_pressure, other_pressure}
OPERATIONS

    pres <-- get_pressure =
    BEGIN
        pres :: PRESSURE
    END
END
```

2.3.2 Create the airlock machine

Create a new machine named `Airlock` to model the airlock door authorization system.

The SEES Clause

In B, the SEES clause is used to **import and reference other machines**. It allows you to **access the sets, constants, variables, and operations** defined in the imported machine **without modifying them**.

To use the **Airlock_pressure_bs basic machine** in your airlock machine, add the following SEES clause in the Airlock machine:

```
SEES
    Airlock_pressure_bs
```

Create the PRESSURE set

Now it's your turn to define the PHASE set for the airlock system. According to the natural language specification, the airlock operates in two distinct phases:

- **ACQ**: Acquisition phase, where the system reads the current pressure.
- **CTRL**: Control phase, where the system decides which doors can be opened.

Introduce an enumerated set named PHASE with the two possible states: ACQ and CTRL. You will use this set to type the cycle variable and control the operation flow in your model.

The VARIABLES clause

An abstract variable in B represents a data element of any type, which may be further refined during component refinement. When introducing variables into your machine, assign a name to each one of them, define its type, optionally specify constraints, and initialize it. To do this, you may add three clauses to your machine: VARIABLES (equivalent to ABSTRACT_VARIABLES), INVARIANT, and INITIALISATION.

Declare the variable names for the airlock door authorization system under the VARIABLES clause, separating them with commas (,):

```
VARIABLES
    airlock_pressure,
    is_indoor_door_openable,
    is_outdoor_door_openable,
    cycle
```

The INVARIANT clause

Invariant specify all variable types and any necessary constraint properties, expressed as predicates. All abstract variables must be fully typed.

This document is the property of CLEARSY.

```
INVARIANT
    airlock_pressure : PRESSURE &
    is_indoor_door_openable : BOOL &
    is_outdoor_door_openable : BOOL &
    cycle : PHASE &
    /* both doors can't be opened at the same time */
    not (is_indoor_door_openable = TRUE & is_outdoor_door_openable = TRUE)
```

The invariant is a crucial part of the machine. It defines the properties that must always hold true for the system. The invariant must be satisfied at all times. It is established during initialization and must be preserved after every operation that can be executed.

The INITIALISATION clause

The `INITIALISATION` clause defines the initial values of all variables using parallel substitutions. It must establish the invariant-meaning that after initialisation, all conditions specified in the INVARIANT clause must be satisfied.

```
INITIALISATION
    is_indoor_door_openable := FALSE ||
    is_outdoor_door_openable := FALSE ||
    airlock_pressure :: PRESSURE ||
    cycle := ACQ
```

2.3.3 Define the operations

We now proceed to complete the operation `actualize_pressure`. It must update the chamber pressure with a value drawn from the set `PRESSURE` and transitions the cycle to `CTRL`.

Start by defining the **precondition** (between `PRE` and `THEN`), which specifies the conditions under which the operation is allowed to execute. In this case, the operation is valid only during the `ACQ` phase:

```
actualize_pressure =
PRE
    cycle = ACQ
```

Next, define the substitutions between `THEN` and `END`. The pressure is nondeterministic assigned a value from `PRESSURE` (using `::`), and the cycle is updated to `CTRL`:

This document is the property of CLEARSY.

```
THEN
    airlock_pressure :: PRESSURE ||
    cycle := CTRL
END;
```

Note that multiple substitutions can be combined in an operation by separating them with ||. These substitutions occur simultaneously, meaning there is no intrinsic order, and all variables refer to their values at the time the operation is invoked.

Then complete the `enable_doors_opening` operation. This operation is responsible for determining whether the inner or outer door can be opened based on the current pressure value.

The operation must be defined as follows:

```
enable_doors_opening =
PRE
    cycle = CTRL
THEN
    is_indoor_door_openable, is_outdoor_door_openable: (
        is_indoor_door_openable: BOOL &
        is_outdoor_door_openable: BOOL &
        (is_indoor_door_openable = TRUE => airlock_pressure = indoor_pressure) &
        (is_outdoor_door_openable = TRUE => airlock_pressure = outdoor_pressure)
    ) ||
    cycle := ACQ
END;
```

We explicitly specify the case where both `is_indoor_door_openable` and `is_outdoor_door_openable` are set to TRUE.

For testing purposes you are going to add getter operations to retrieve the values of `airlock_pressure`, `is_indoor_door_openable`, `is_outdoor_door_openable` and `cycle`variables. These operations will allow you to check the current state of the airlock system during test with C code.

```
    ret <-- get_airlock_pressure =
    PRE
        ret : PRESSURE
    THEN
```


This document is the property of CLEARSY.

```
    ret := airlock_pressure
END;

ret <-- get_is_indoor_door_openable=
PRE
    ret : BOOL
THEN
    ret := is_indoor_door_openable
END;

ret <-- get_is_outdoor_door_openable=
PRE
    ret : BOOL
THEN
    ret := is_outdoor_door_openable
END;

ret <-- get_cycle =
PRE
    ret : PHASE
THEN
    ret := cycle
END
```

Save the machine. **Atelier B's simple prover** of the main panel may not be able to prove the proof obligation (PO) automatically. To proceed with a better prover, follow these steps:

1. **Select the machine** and click the  button to open the **interactive prover panel**.
2. In the **middle-left panel**, select your PO. It will appear in the central panel.
3. Click the  button to attempt the proof manually.
4. Once the PO is proved, its status will change to **Proved** and the screen will turn **green**.
5. To exit, click the **X** button in the top-right corner of the interactive prover panel.
6. When prompted to save your proof in the **User Pass**, click **Yes** > **Next** > **Finish**. Back in the main view, you should now see the proof obligation (PO) of your airlock machine is now proved.

This document is the property of CLEARSY.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
Airlock	OK	OK	1	1	0	OK
Airlock_pressure_bs	OK	OK	0	0	0	OK

Figure 12: Airlock machine proof obligations

2.4 Animation with ProB

To validate the behavior of the airlock authorization system, you use ProB, a model checker and animator for B machines. ProB allows you to simulate the behavior of your B models, check properties, and visualize the state of your system.

Click on the ProB2-UI application to launch it. You should see a window similar to the one below:

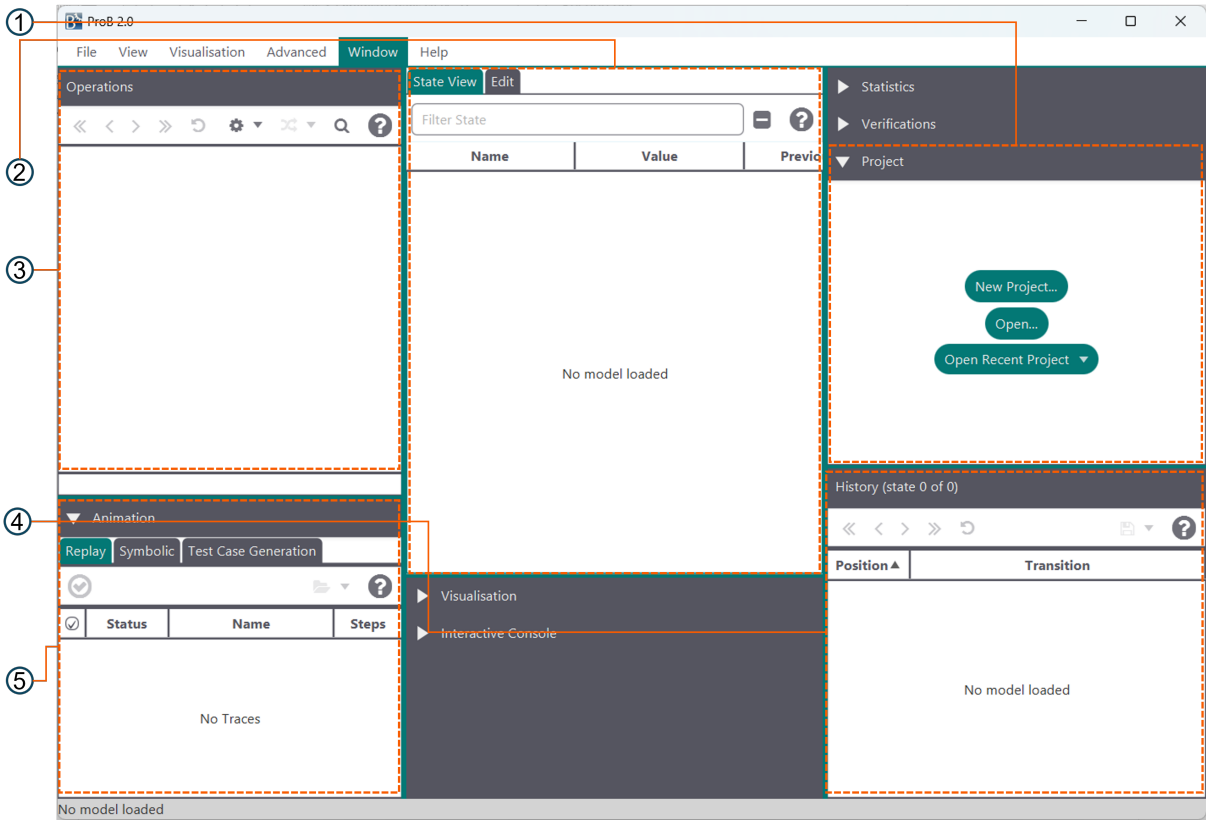


Figure 13: Main Window of ProB

1. The project section in the **right panel** shows a list of available projects and machines you have imported or worked on.

2. The **center panel** displays the current state of the loaded machine, including constants, variables, alongside their values. You can either see or edit its source code by using the Edit button at the top of the center panel.
3. The **top-left panel** shows the list of available operations for the currently loaded machine. Operations that can be executed from the current state will be shown in green. You can left-click on any operation to run it. If you right-click, you can enter specific parameters or add constraints before running the operation.
4. The **bottom-right panel** logs the execution trace of operations, which keeps a history of all operations you have performed. This lets you step back through previous states to review what happened.
5. The **bottom-left panel** allows you to replay previously saved execution traces to analyze past executions or reproduce specific scenarios.

To use ProB2-UI, you first need to load your B machine. You can do this by importing the `.mch` file of your machine into ProB2-UI.

1. Go to the menu **File** > **Open**.
2. In the file dialog, navigate to your **Atelier B** project directory and select the `Airlock.mch` machine file. Ensure both the airlock machine `Airlock` and the basic machine `Airlock_pressure_bs` are in the same folder.

Once the machine is loaded, you can see on the top left panel the list of available operations for the airlock machine. Since the "airlock_pressure" is non-deterministically initialized, three INITIALISATION states are possible one for each value of the PRESSURE set (`indoor_pressure`, `outdoor_pressure`, `other_pressure`). Choose one of the available `INITIALISATION` operations to begin the animation. This will initialize all the machine's variables.

To see the current state of the machine, look at the **center panel** in the 'State View' section. You should see the initial values of all variables, including `airlock_pressure`, `is_indoor_door_openable`, `is_outdoor_door_openable`, and `cycle`.

You can now interact with the machine by running operations. The available operations are listed in the **top-left panel**. You can run them by clicking on them. In ProB2-UI, operations are marked with different icons to indicate their effect on the system state:

- A **green arrow** means the operation modifies the state by changing the value of at least one variable.
- A **circular blue arrow** means the operation does not change the state.

You can test the airlock authorization system by following this sequence of operations:

This document is the property of CLEARSY.

1. **Run the `actualize_pressure` operation**:
   This simulates acquiring the current pressure of the airlock chamber. The pressure will be set to a nondeterministic value from the set `PRESSURE`, and the cycle will transition to `CTRL`.
2. **Run the `enable_doors_opening` operation**: This operation evaluates the current pressure and determines whether the inner or outer door can be opened based on the pressure value acquired in the previous step. The cycle will then transition back to `ACQ`.
3. **Repeat the above two operations**: Continue alternating between `actualize_pressure` and `enable_doors_opening` to simulate the airlock system's operation over time.

Observe the changes in the state of the machine as you perform these operations. You should see the values of `airlock_pressure`, `is_indoor_door_openable`, and `is_outdoor_door_openable` change according to the rules defined in your model.

1. **Check Variable Values:**
   Run the operations and observe the values of `airlock_pressure`, `is_indoor_door_openable`, and `is_outdoor_door_openable` in the state view. Ensure that these variables change as expected after each operation.
2. **Verify Invariant Maintenance:**
   After each operation, confirm that the invariant is preserved: both `is_indoor_door_openable` and `is_outdoor_door_openable` should never be TRUE at the same time. They must always be either TRUE or FALSE, but not both TRUE simultaneously.
3. **Observe Operation Cycling:**
   Execute the `actualize_pressure` and `enable_doors_opening` operations in sequence. Confirm that the system alternates between these two phases during animation, reflecting the intended control cycle.

Model checking with ProB2-UI helps verify that the system behaves as specified, but it does not prove the model is correct. Correctness also depends on whether the specifications reflect real-world requirements. Formal proof of the generated code is handled by Atelier B and the B Method, which ensure that the implementation is mathematically consistent with the specification.

2.5 Implementation

After completing the specification of the abstract airlock model and its operations, the next step is to develop an implementation model based on this specification.

This document is the property of CLEARSY.

Create a new implementation model to implement the airlock authorization system. This implementation will refine the abstract model and provide concrete definitions for the variables and operations.

In the implementation, all abstract variables used in the abstract model must be explicitly declared under the `CONCRETE_VARIABLES` clause:

```
CONCRETE_VARIABLES
    airlock_pressure,
    cycle,
    is_indoor_door_openable,
    is_outdoor_door_openable
```

You do **not** need to repeat the `INVARIANT` clause here, as it is automatically inherited from the abstract machine.

The `INITIALISATION` clause must be explicitly defined, as it assigns concrete initial values to all variables in the system. This allows the invariant to hold from the start of execution. Unlike in the abstract machine, where types such as `PRESSURE` can remain in the `other_pressure` state, the implementation requires each variable to be assigned a precise value from its corresponding set.

A suitable and secure initial value for `airlock_pressure` is `other_pressure`, since the sensor has not yet performed any readings. This represents a state in which all doors are closed. The system should also begin in the Acquisition (`ACQ`) phase, so it is ready to perform its first pressure scan.

```
INITIALISATION
    airlock_pressure := other_pressure;
    is_indoor_door_openable := FALSE;
    is_outdoor_door_openable := FALSE;
    cycle := ACQ
```

In implementation models, substitutions are executed sequentially in the same manner as a common programming language. Furthermore, sequential substitutions are separated by a ';' instead of '||'. In B, you must not place a (';') after the last statement.

The `actualize_airlock_pressure` operation is responsible for updating the airlock pressure. To use the sensor, you need to import the operation from the basic machine `Airlock_pressure_bs` you created earlier.

This document is the property of CLEARSY.

```
SEES
    Airlock_pressure_bs
```

This operation retrieves the current pressure from the sensor, changes the value of `airlock_pressure`, and the cycle status is transited to `CTRL` by the operation.

The B implementation of this operation is as follows:

```
actualize_pressure =
BEGIN
    airlock_pressure <-- get_pressure ;
    cycle := CTRL
END;
```

The next operation to implement is `enable_doors_opening`. This operation determines whether the inner or outer door can be opened based on the current airlock pressure. It updates the door permission variables accordingly and then resets the system phase back to acquisition (`ACQ`) to prepare for the next pressure reading.

You may use a `CASE` substitution to evaluate the current pressure and assign the appropriate Boolean value to `is_indoor_door_openable` and `is_outdoor_door_openable`. You must not forget to set the cycle variable `ACQ` at the end:

```
enable_doors_opening =
BEGIN
    CASE airlock_pressure OF
        EITHER indoor_pressure THEN
            is_indoor_door_openable := TRUE;
            is_outdoor_door_openable := FALSE
        OR outdoor_pressure THEN
            is_indoor_door_openable := FALSE;
            is_outdoor_door_openable := TRUE
        ELSE
            is_indoor_door_openable := FALSE;
            is_outdoor_door_openable := FALSE
        END
    END;
    cycle := ACQ
END;
```


This document is the property of CLEARSY.

With both operations completed, your airlock authorization system now follows a full acquisition-control cycle.

To facilitate observation and verification of the system's state when running the generated C code, implement getter operations that return the current values of the airlock pressure and door authorization variables.

```
ret <-- get_airlock_pressure =
BEGIN
    ret := airlock_pressure
END;

ret <-- get_is_indoor_door_openable =
BEGIN
    ret := is_indoor_door_openable
END;

ret <-- get_is_outdoor_door_openable =
BEGIN
    ret := is_outdoor_door_openable
END;

ret <-- get_cycle =
BEGIN
    ret := cycle
END
```

2.6 Proving the implementation

There are no proof obligations to prove for `Airlock_i`. Atelier B does not generate additional proof obligations for this component. After saving and type-checking, you should see "Typechecked" and "POs Generated" with "OK" status, and zero proof obligations in the main view.

2.7 Generating and testing the C code

Select your machines **Component** > **B0 Check** to verify that the implementation is valid and consistent with the abstract specification and is ready for code generation.

Here is what you should see in the main view:

This document is the property of CLEARSY.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
Airlock	OK	OK	1	1	0	OK
Airlock_i	OK	OK	0	0	0	OK
Airlock_pressure_bs	OK	OK	0	0	0	OK

Figure 14: Airlock B0 Check

Once the implementation is B0 checked, you can generate the associated C code using the built-in translator.

To generate the associated C code, select all implementations machines (you can use Shift-click to select multiple files). In this case select only the `Airlock_i` machine. Then, right-click on the selection, and choose **Code generator**. In the dialog, set the language to **C** and the profile to **C9X**, then click **OK**. This will generate C files for all components in your project. For the `Airlock_i` machine, this will generate the following C files: **`Airlock_i.c`**, **`Airlock.h`**.

Repeat the process for each basic machine to generate the associated C code for the basic machine. In this case, select only the `Airlock_pressure_bs` machine. This will generate the C file **`Airlock_pressure_bs.h`**.

To access all generated files, right-click your project in the workspaces tab and select **Open folder**. The C files will be located in the `lang/c` subdirectory of your B project.

A basic machine models data sources that cannot be formally proven correct, such as physical sensors. While the B Method ensures the correctness of the logic that uses sensor data, it cannot guarantee the reliability of the sensor itself. Therefore, the value returned by the sensor must be provided by external code, and the overall system's safety is achieved through redundancy (e.g., multiple sensors) and error detection mechanisms.

For this example, you need to implement the pressure sensor in C as defined by the basic machine `Airlock_pressure_bs`. Specifically, provide the `get_pressure` operation, which simulates reading the current pressure from a sensor. For simplicity, this function can randomly select a value from the `PRESSURE` set.

Create a new file named `Airlock_pressure_bs.c` with the following content:

```c
/* File: Airlock_pressure_bs.c */
#include "Airlock_pressure_bs.h"
#include <stdlib.h>

void Airlock_pressure_bs__get_pressure(Airlock_pressure_bs__PRESSURE *pres) {
    int random_value = rand() % Airlock_pressure_bs__PRESSURE__max;
```

```
    switch (random_value) {
        case 0:
            *pres = Airlock_pressure_bs__indoor_pressure;
            break;
        case 1:
            *pres = Airlock_pressure_bs__outdoor_pressure;
            break;
        default:
            *pres = Airlock_pressure_bs__other_pressure;
            break;
    }
}
```

This code uses the `rand()` function to generate a random number, which is then mapped to one of the three pressure values defined in the `PRESSURE` set.

This implementation provides a simulated sensor value. In real projects, you must ensure the reliability of your sensor through hardware validation, redundancy, and thorough testing.

To run the code, you need to create a new file (for example, `main.c`) containing the following main function. Normally, this file also needs to be proven or validated as part of your project:

```
/* File: main.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "Airlock.h"
#include "Airlock_pressure_bs.h"

int main(void) {
    // Seed the random number generator
    srand((unsigned int)time(NULL));

    // Initialize the modules
    Airlock__INITIALISATION();

    // Variables to read the results
    Airlock_pressure_bs__PRESSURE pressure;
    bool indoor_openable;
    bool outdoor_openable;
    Airlock__PHASE phase;
```


This document is the property of CLEARSY.

```
// Loop to simulate several cycles
for (int i = 0; i < 10; i++) {
    printf("Cycle %d :\n", i + 1);



    // Step 1: Get new pressure value
    Airlock__actualize_pressure();

    Airlock__get_cycle(&phase);
    printf("Cycle Phase: %s\n\n", (phase == Airlock__ACQ) ? "ACQ" : "CTRL");

    // Step 2: Enable door opening based on the pressure
    Airlock__enable_doors_opening();



    Airlock__get_airlock_pressure(&pressure);
    Airlock__get_is_indoor_door_openable(&indoor_openable);
    Airlock__get_is_outdoor_door_openable(&outdoor_openable);
    Airlock__get_cycle(&phase);

    // Display results
    const char* pressure_str;
    switch (pressure) {
        case Airlock_pressure_bs__indoor_pressure:
            pressure_str = "Indoor Pressure";
            break;
        case Airlock_pressure_bs__outdoor_pressure:
            pressure_str = "Outdoor Pressure";
            break;
        case Airlock_pressure_bs__other_pressure:
        default:
            pressure_str = "Other Pressure";
            break;
    }

    printf("Pressure: %s\n", pressure_str);
    printf("Indoor Door Openable: %s\n", indoor_openable ? "Yes" : "No");
    printf("Outdoor Door Openable: %s\n", outdoor_openable ? "Yes" : "No");
    printf("Cycle Phase: %s\n\n", (phase == Airlock__ACQ) ? "ACQ" : "CTRL");
```


This document is the property of CLEARSY.

```
    }

    return 0;
}
```

This `main.c` file initializes the airlock system, simulates several cycles of pressure reading and door authorization, and prints the results to the console. It uses the functions generated from your B machine to interact with the airlock system.

Make sure to include the necessary headers and link against the generated C files when compiling your project. You can use this `Makefile` similar to the one provided in the previous exercise to compile and run your code.

Makefile

```
objects = Airlock_pressure_bs.o Airlock_i.o main.o

all: $(objects)
    $(CC) $^ -o sas

%.o: %.c
    $(CC) -c $< -o $@

clean:
    rm -f *.o sas
```

You should have the following files in your folder before running the Makefile:

Generated files:

- `Airlock_i.c`
- `Airlock_i.h`
- `Airlock_pressure_bs.h`

Created files: * `Airlock_pressure_bs.c` (your sensor simulation)

- `main.c`
- `Makefile` (the build script)

After ensuring all required files are in place, simply run the provided Makefile to build the project:

This document is the property of CLEARSY.

```
make
```

This will produce an executable named sas. You can then run it to test the airlock authorization system:

```
./sas
```

Here is a trace of the expected output:

```
Cycle 1 :
Cycle Phase: CTRL

Pressure: Other Pressure
Indoor Door Openable: No
Outdoor Door Openable: No
Cycle Phase: ACQ

Cycle 2 :
Cycle Phase: CTRL

Pressure: Indoor Pressure
Indoor Door Openable: Yes
Outdoor Door Openable: No
Cycle Phase: ACQ

Cycle 3 :
Cycle Phase: CTRL

Pressure: Indoor Pressure
Indoor Door Openable: Yes
Outdoor Door Openable: No
Cycle Phase: ACQ

Cycle 4 :
Cycle Phase: CTRL

Pressure: Outdoor Pressure
Indoor Door Openable: No
Outdoor Door Openable: Yes
Cycle Phase: ACQ
```

```
Cycle 5 :
Cycle Phase: CTRL

Pressure: Outdoor Pressure
Indoor Door Openable: No
Outdoor Door Openable: Yes
Cycle Phase: ACQ

Cycle 6 :
Cycle Phase: CTRL

Pressure: Outdoor Pressure
Indoor Door Openable: No
Outdoor Door Openable: Yes
Cycle Phase: ACQ

Cycle 7 :
Cycle Phase: CTRL

Pressure: Other Pressure
Indoor Door Openable: No
Outdoor Door Openable: No
Cycle Phase: ACQ

Cycle 8 :
Cycle Phase: CTRL

Pressure: Other Pressure
Indoor Door Openable: No
Outdoor Door Openable: No
Cycle Phase: ACQ

Cycle 9 :
Cycle Phase: CTRL

Pressure: Indoor Pressure
Indoor Door Openable: Yes
Outdoor Door Openable: No
Cycle Phase: ACQ

Cycle 10 :
```


This document is the property of CLEARSY.

```
Cycle Phase: CTRL

Pressure: Indoor Pressure
Indoor Door Openable: Yes
Outdoor Door Openable: No
Cycle Phase: ACQ
```

2.8 To go further

- **Redundant pressure sensors:** Enhance the airlock authorization system by introducing three independent pressure sensors, similar to the redundancy used in the railroad switch example. The system should determine the chamber pressure by majority voting: if at least two sensors agree, that value is used as the chamber pressure. If all three sensors report different values, the system must not authorize either door to open. Update both the B specification and implementation to reflect this majority-based decision logic.

This document is the property of CLEARSY.

3 Integer arithmetic calculator

3.1 Introduction

By completing this exercise, you will:

- Learn how to **declare and initialize integer variables** in B.
- Use **arithmetic operators** (+, −, *, /, mod) to perform calculations with integers.
- Apply **relational operators** (=, /=, <, >, <=, >=) to compare integer values.
- Manipulate **sets of integers** such as INT, NAT.
- Address **bounds and overflow issues** when working with integer values.
- Use **predicates** involving integers in operations and assertions.
- **Verify the correctness** of integer operations using formal proofs in B.
- Make a main machine that uses the integer arithmetic machine to perform basic calculations and demonstrate the use of integers in B.

3.2 Natural language description

In this exercise, you are asked to specify a simple machine that can perform basic arithmetic operations on integers. The machine will manage three variables: aa, bb, and result. The variables aa and bb will store integer values, while result will hold the outcome of arithmetic operations performed using aa and bb.

The machine should provide operations to set the values of aa and bb individually. Once these values are set, the machine must be able to perform various arithmetic operations, including addition, subtraction, multiplication, division, modulus (remainder), and exponentiation. Each operation should store its result in the result variable.

Additionally, the machine should include an operation to determine the maximum of the two integers (aa and bb) and store this value in result. There should also be a getter operation to retrieve the current value of result.

To demonstrate the use of this arithmetic machine, you will also specify a main machine. This main machine will import the arithmetic machine and use its operations to perform example calculations, showcasing how integer arithmetic can be modeled and used in B.

3.3 Formal specification

To begin, create a new project named Integer_arithmetic in Atelier B. Make sure to set the INT type to a 32-bit signed integer.

This document is the property of CLEARSY.

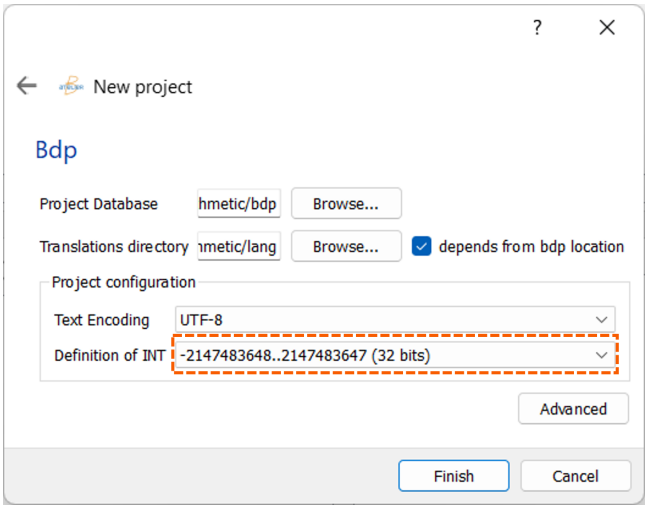


Figure 15: Atelier B project settings

Next, create an abstract B machine named `Integer_arithmetic`. This machine will define the basic arithmetic operations on integers, including addition, subtraction, multiplication, division, modulus, exponentiation, and finding the maximum of two integers.

```
MACHINE
    Integer_arithmetic
VARIABLES
    aa, bb, result
INVARIANT
    aa : INT &
    bb : INT &
    result : INT
INITIALISATION
    aa :: INT ||
    bb :: INT ||
    result :: INT
OPERATIONS

    /* Setters */
    set_a(xx) =
    PRE xx : INT THEN
        aa : (aa = xx)
    END;

    set_b(xx) =
    PRE xx : INT THEN
```


This document is the property of CLEARSY.

```
    bb : (bb = xx)
END;

/* Getters */
res <-- get_result =
PRE result : INT THEN
    res : (res = result)
END;

/* Operations */
sum_ab =
PRE aa + bb : INT THEN
    result : (result = aa + bb)
END;

sub_ab =
PRE aa - bb : INT THEN
    result : (result = aa - bb)
END;

mul_ab =
PRE aa * bb : INT THEN
    result : (result = aa * bb)
END;

div_ab =
PRE not(bb = 0) & aa / bb : INT THEN
    result : (result = aa / bb)
END;

mod_ab =
PRE aa >= 0 & bb >= 1 & aa mod bb : INT THEN
    result : (result = aa mod bb)
END;

pow_a(pp) =
PRE pp : NAT & aa**pp : INT THEN
    result : (result = aa ** pp)
END;

max_ab =
BEGIN
```


This document is the property of CLEARSY.

```
        result := max({aa, bb})
    END
END
```

3.3.1 Specification overview

The variables **aa**, **bb**, and **result** are declared of type INT. **INT** is a predefined type that represents a finite subset of the set of **INTEGERS** ($\mathbb{Z}$) and it is defined as **MININT..MAXINT**, where **MININT** and **MAXINT** are constants that specify the minimum and maximum values for the implementable integers, respectively. They are defined at the creation of the project. In this case, they are set to the limits of a 32-bit signed integer, where **MININT = -2147483648** and **MAXINT = 2147483647**.

In the operation **pow_a**, the exponent **pp** is declared as a NAT. **NAT** is another predefined type that represents a finite subset of the set of **NATURAL** ($\mathbb{N}$), defined as **0..MAXINT**.

Select the first PO in the **INVARIANT** section. Because **result** are of type **INT**, each operation must confirm that the results of the arithmetic operations are also integers. With a precondition, you specify that the result of the operation is valid only if the computed value belongs to INT. For example, the addition operation **sum_ab** has a precondition that checks if **aa + bb** is an integer.

The operation for division (**div_ab**) and for modulus (**mod_ab**) need additional preconditions to ensure their validity:

- **Division**: The precondition checks that **bb** is not zero.
- **Modulus**: The precondition checks that **aa** is non-negative and **bb** is a positive integer.

Remove the preconditions `not(bb = 0)` from the **div_ab** operation. Then select the associated PO in the **OPERATIONS** section. The operation is not defined in this case, so we need to ensure that the operations are only called with valid parameters. These preconditions are known as **well-definedness conditions**.

The precondition PRE ensures that the division operation is only performed when **bb** is not zero, preventing division by zero errors. You can put back the precondition `not(bb = 0)` in the **div_ab** operation.

It is the same for the **mod_ab** operation that has a precondition that checks that **aa** is non-negative and **bb** is a positive integer. This ensures that the modulus operation is valid and avoids errors when performing the operation.

The **exponentiation** operation (**pow_a**) is of type **INT × NAT → INT**, meaning it takes an integer **aa** and a natural number **pp** as parameters.

This document is the property of CLEARSY.

The **max_ab** operation uses the **max** arithmetic expression, which returns the maximum value of a set. **{aa, bb}** create the set containing the two integers **aa** and **bb**.

You can now save the machine, it should prove automatically.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
Integer_arithmetic	OK	OK	17	17	0	-

Figure 16: Integer arithmetic machine proof obligations

To demonstrate the use of the **Integer_arithmetic** machine, create a **Main** machine that will imports and calls its operations to perform calculations.

Within Atelier B, a **Main** machine is composed of a single operation and does not declare any variables. This operation, typically named **main**, takes no parameters and returns no value. Use the **skip** substitution as you will implement the main operation later.

```
MACHINE
    Main
OPERATIONS
    main = skip
END
```

3.4 Implementation

Now that you have defined the abstract machine `Integer_arithmetic` with its operations, the next step is to create an implementation model that refines this abstract machine.

Create a new implementation model for `Integer_arithmetic`.

```
IMPLEMENTATION Integer_arithmetic_i
REFINES Integer_arithmetic

CONCRETE_VARIABLES
    aa, bb, result

INITIALISATION
    aa := 0 ;
    bb := 0 ;
    result := 0
```


This document is the property of CLEARSY.

```
OPERATIONS
    set_a(xx) =
    BEGIN
        aa := xx
    END;


    set_b(xx) =
    BEGIN
        bb := xx
    END;

    res <-- get_result =
    BEGIN
        res := result
    END;

    /* Operations */

    sum_ab =
    BEGIN
        result := aa + bb
    END;


    sub_ab =
    BEGIN
        result := aa - bb
    END;


    mul_ab =
    BEGIN
        result := aa * bb
    END;


    div_ab =
    BEGIN
        result := aa / bb
    END;
```


This document is the property of CLEARSY.

```
    mod_ab =
    BEGIN
        result := aa mod bb
    END;


    pow_a(pp) =
    BEGIN
        result := aa ** pp
    END;

    max_ab =
    BEGIN
        IF aa >= bb THEN
            result := aa
        ELSE
            result := bb
        END
    END

END
```

As you can see, preconditions in the abstract machine are not present in the implementation model. The specification guarantees that the operations will only be called with valid parameters, so the implementation does not need to be checked there but when calling the operations in the main machine.

You can now save the implementation model, it should prove automatically.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
Integer_arithmetic	OK	OK	17	17	0	-
Integer_arithmetic_i	OK	OK	10	10	0	-
Main	OK	OK	0	0	0	-

Figure 17: Integer arithmetic implementation model proof obligations

Then create an implementation of the main machine that will use the `Integer_arithmetic` machine to perform basic calculations. This implementation will refine the abstract `Main` machine and provide concrete definitions for the operations.

This document is the property of CLEARSY.

```
IMPLEMENTATION Main_i
REFINES Main

IMPORTS
    Integer_arithmetic
OPERATIONS
    main =
    VAR aa, bb, res, pp IN
        aa := 5;
        bb := 2 ;
        res := 0;
        pp := 1;

        set_a(aa) ;
        set_b(bb) ;

        sum_ab ;
        res <-- get_result;

        sub_ab ;
        res <-- get_result ;

        mul_ab ;
        res <-- get_result;

        div_ab ;
        res <-- get_result ;

        mod_ab ;
        res <-- get_result ;

        max_ab ;
        res <-- get_result ;

        pow_a(pp) ;
        res <-- get_result
    END
END
```

Save the implementation model. To prove the implementation model, click on the  button in the main view.

This document is the property of CLEARSY.

To thoroughly test the preconditions of each arithmetic operation, experiment by assigning various values to the variables `aa`, `bb`, and `pp` in the `main` implementation.

Try assigning `MAXINT` to `aa` and setting `bb := 2`. The addition operation will fail to prove, as it cannot guarantee that `MAXINT + 2` is still an integer within the allowed range. Similarly, if you assign `MININT` to `aa`, the subtraction operation will not be provable because it cannot ensure that `MININT - 2` remains a valid integer.

In an operation of an implementation, once a PO is proved false, all subsequent POs will be considered wrongly as correct because each substition is executed sequentially and has condition that all previous substitutions have been executed successfully, which is false in this case. To observe the effect of the initialisation on other preconditions, you can comment out the operations in the main implementation model and uncomment them one by one to see how the model checks the preconditions.

Keep the variable `aa` set to `MAXINT` and `bb` set to 2. Comment the addition and subtraction operations in the main implementation model. You will see that the multiplication operation is also not provable because the multiplication of two integers can also overflow.

You can also test the division and modulus operations by setting `bb` to zero. The division operation will fail to prove because the precondition requires `bb` to be non-zero. The modulus operation will also fail to prove because it requires `bb` to be a positive integer.

Finally, test the exponentiation operation by setting `pp` to a negative value. The precondition requires `pp` to be a natural number, so if you set `pp` to `-1`, the exponentiation operation will also fail to prove.

Reinitialize all the variables respectively to `aa := 5`, `bb := 2`, and `pp := 2`. Uncomment the addition, subtraction, multiplication, division, modulus, and exponentiation operations in the main implementation model. Then click on the  button in the main view to prove the implementation model.

The automatic prover will now be able to discharge all proof obligations for the operations, except for the preconditions related to the exponentiation operation. To assist the prover, you need to provide an explicit proof for this case by adding a proof file to your project:

1. Open your project folder by right-clicking the project name in the main view and selecting **Open folder**.
2. Create a new file named `Main_i.pmm` in the project directory. This file will contain the user proof for the implementation model.

Add the following content to `Main_i.pmm`:

This document is the property of CLEARSY.

Main_i.pmm

```
THEORY User_Pass IS
Operation(Operation_main) & ff(0) & dd(0) & eh(aa) & ar(MyPowerT.1,Goal) & mp
END &
THEORY MyPowerT IS
5**2 == 5*5
END
```

To apply your user proof and complete the proof, click the  button in the proof obligations panel. This will load and apply the contents of `Main_i.pmm`, allowing Atelier B to discharge the remaining proof obligations automatically.

The implementation model proves the code to be coherent with the implementation. The implementation model ensures that the code is consistent with the abstract specification, meaning there are no errors in the code that would prevent the implementation from fulfilling the operations defined in the abstract machine.

Select all machine and use `CTRL + B` to B0 check all components.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
Main_i	OK	OK	19	19	0	OK
Integer_arithmetic	OK	OK	17	17	0	OK
Integer_arithmetic_i	OK	OK	10	10	0	OK
Main	OK	OK	0	0	0	OK

Figure 18: Integer arithmetic main implementation model proof obligations

3.5 Generating C code

You can now generate the C code for the implementation model. Right-click on the `Main_i` machine in the main view and select **Component** > **Code generator**. This will generate the C code for the implementation model. Tick the option to create a main file. The generated C code will include the necessary headers and function definitions for the operations defined in the `Integer_arithmetic_i` machine. The main function will call the `main` operation to perform the calculations.

To display the results of the operations, modify the generated `Main_i.c` file to add the following `printf` statements to display the results of the operations.

This document is the property of CLEARSY.

Never modify code that has already been proved, as this invalidates the formal proof and the guarantees provided by the B Method. The code modifications shown here are solely for testing and display purposes. In practice, you should NEVER alter proved code outside the formal development process.

```c
/* File: Main.c */
#include <stdio.h>

#include "Main.h"

/* Clause IMPORTS */
#include "Integer_arithmetic.h"

/* Clause CONCRETE_CONSTANTS */
/* Basic constants */

/* Array and record constants */
/* Clause CONCRETE_VARIABLES */

/* Clause INITIALISATION */
void Main__INITIALISATION(void)
{

    Integer_arithmetic__INITIALISATION();
}

/* Clause OPERATIONS */

void Main__main(void)
{
    int32_t aa;
    int32_t bb;
    int32_t res;
    int32_t pp;

    aa = 5;
    bb = 2;
    res = 35;
    pp = 2;
    printf("aa = %d\n", aa);
    printf("bb = %d\n", bb);
    printf("pp = %d\n", pp);
```


This document is the property of CLEARSY.

```
    Integer_arithmetic__set_a(aa);
    Integer_arithmetic__set_b(bb);

    Integer_arithmetic__sum_ab();
    Integer_arithmetic__get_result(&res);
    printf("sum_ab: %d\n", res);

    Integer_arithmetic__sub_ab();
    Integer_arithmetic__get_result(&res);
    printf("sub_ab: %d\n", res);

    Integer_arithmetic__mul_ab();
    Integer_arithmetic__get_result(&res);
    printf("mul_ab: %d\n", res);

    Integer_arithmetic__mod_ab();
    Integer_arithmetic__get_result(&res);
    printf("mod_ab: %d\n", res);

    Integer_arithmetic__max_ab();
    Integer_arithmetic__get_result(&res);
    printf("max_ab: %d\n", res);

    Integer_arithmetic__pow_a(pp);
    Integer_arithmetic__get_result(&res);
    printf("pow_a: %d\n", res);
}
```

Use the following makefile to compile the generated C code, with the command `make all`:

```
objects = b_pow.o Integer_arithmetic_i.o Main_i.o b_main.o
all: $(objects)
    $(CC) $^ -o calculator

$(objects): %.o: %.c
    $(CC) -c $^ -o $@

clean:
    rm -f *.o calculator
```

Here is a trace of the expected output when running the generated code:

This document is the property of CLEARSY.

```
aa = 5
bb = 2
res = 35
pp = 2
sum_ab: 7
sub_ab: 3
mul_ab: 10
mod_ab: 1
max_ab: 5
pow_a: 25
```

3.6 To go further

You can extend the `Integer_arithmetic` machine by adding more operations, such as:

- **Square root**: Implement an operation to calculate the square root of a non-negative integer.
- **Absolute value**: Implement an operation to calculate the absolute value of an integer.
- **GCD (Greatest Common Divisor)**: Implement an operation to calculate the GCD of two integers.
- **LCM (Least Common Multiple)**: Implement an operation to calculate the LCM of two integers.

This document is the property of CLEARSY.

4 Fuel level

4.1 Introduction

The B Method's **modular architecture** notion is introduced in this practice. Ensuring safety-critical properties, such as accurate fuel level estimation, is essential in embedded systems like aviation, where errors can have catastrophic consequences.

To explore this concept, we will model a system that estimates the remaining amount of fuel in a tank using redundant sensors and raises an alarm when the estimated fuel level drops below a critical threshold. The system integrates data from two ultrasonic level sensors and three flowmeters, combining their readings to improve accuracy and reliability despite potential sensor errors.

Through this exercise, you will learn about:

- **Modular architecture**: The best practices on how to divide your B code into multiple interconnected machines, giving a hand to the automatic prover and factorizing the code.
- **Practice your B skills**: Specify conditions under which the system must alert the pilot about low fuel levels while working with redundancy and Boolean logic.

Designing such a model highlights challenges in handling uncertainty and fault tolerance, crucial for developing dependable safety-critical software.

4.2 Natural language description

In this exercise, we consider a safety-critical system responsible for triggering an alarm when the remaining amount of fuel in an aircraft's tank in bellow a given threshold. A pessimistic fuel estimation is essential, as an incorrect value could result in fuel exhaustion mid-flight, potentially leading to catastrophic failure.

To address this challenge, the system integrates redundant sensing mechanisms to improve robustness against faulty or inaccurate sensor readings. The fuel estimation process relies on two types of sensors:

- Two ultrasonic level sensors, which measure the initial fuel level in the tank.
- Three flowmeters, which measure the amount of fuel consumed over each cycle.

Because sensor data can be noisy or unreliable, the system applies conservative fusion strategies to ensure safety:

- At system startup, the initial fuel level is computed as the minimum of the two ultrasonic measurements.

This document is the property of CLEARSY.

- During operation, fuel consumption is estimated at each cycle as the maximum value reported by the three flowmeters. This value is subtracted from the previously estimated fuel level.

Since we are dealing with pessimistic estimations, we always consider the scenario with the highest fuel consumption and the minimum fuel level between the different values returned by the sensors. For safety purposes, the fuel level estimation is attended to be lower than the actual level.

The system continuously updates the estimate of the remaining fuel based on this conservative model. When the estimated level falls below or equal 30%, the predefined threshold, an alarm is raised to inform the pilot of a potential fuel shortage. This particular aircraft's tank has a maximum capacity of 1000 liters, so whenever the fuel level is bellow 300 liters, an alarm must be triggered.

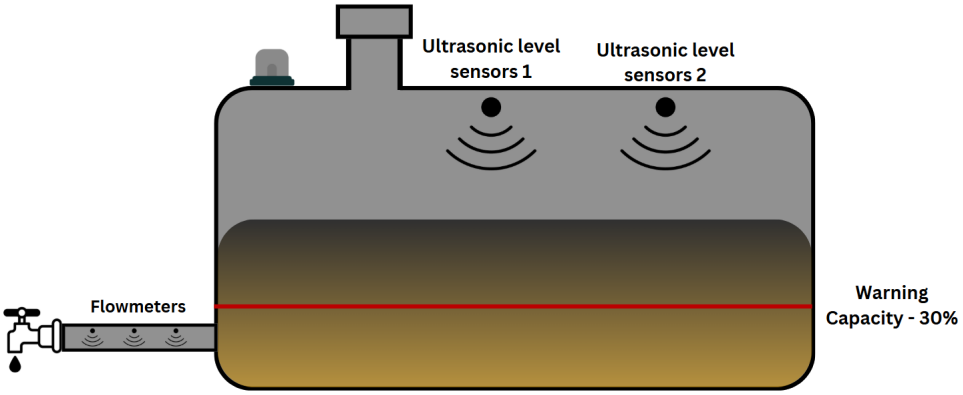


Figure 19: Example of a fuel level where the alarm is not active

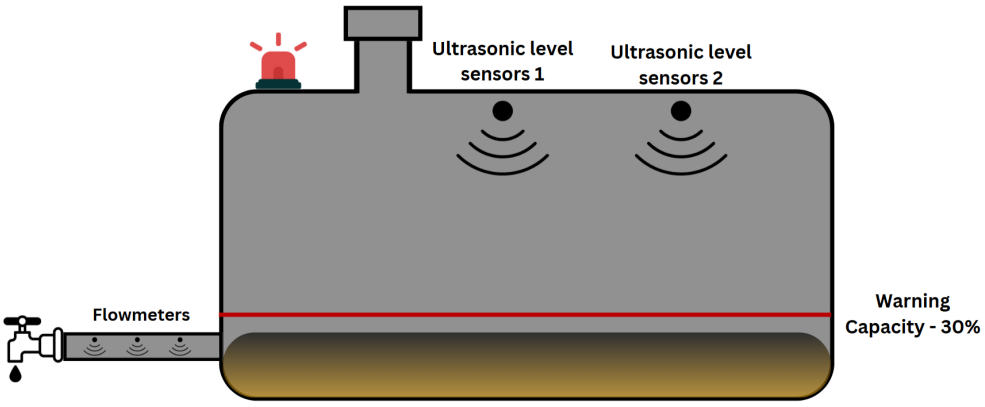


Figure 20: Example of a fuel level where the alarm is active

This document is the property of CLEARSY.

In both images, the fuel level is represented as a percentage of the tank's total capacity. The first image shows a fuel level above 30%, where the alarm is not active, while the second image shows a fuel level below or equal to 30%, where the alarm is triggered. We can also see the two ultrasonic sensors and three flowmeters, which are used to estimate the fuel level.

The exercise consists in constructing this aircraft's fuel level alarm through B modeling, particularly focusing on ensuring that fuel shortage alarms are never missed, even in the presence of sensor faults.

You must use modular architecture to structure your B code into multiple interconnected machines. This approach will help to prove the code, and improve code organization, thus augmenting its maintainability.

4.3 Modular architecture

While working in more complex projects, modular architecture is a good practice that helps to organize the code into smaller, more manageable components. In B, this is achieved by dividing the code into multiple machines, each responsible for a specific part of the system.

The major advantage of modular architecture is that B variables cannot be implemented in more than one component. Importing allows the algorithm to be decomposed into components that handle independent sets of variables.

This factorization also helps prevent circular imports. **Circular imports** occur when two or more implementations models depend on each other by importing each other directly or indirectly. For example, if implementation model B imports implementation model A, and implementation model A also imports something from implementation model B, this creates a dependency loop. In many programming languages, such as B, this can lead to errors or unpredictable behavior because one module may try to access code that hasn't been fully loaded or defined yet.

These are the rules to properly structure B code:

- It is considered a good practice to not mix constants/sets and variables. Create a separate machine for constants and sets, which can be accessed from other machines using the `SEES` clause.
- Utility operations, which use no variables besides those passed as parameters, should be placed in a separate machine. Utility operations can, for example, consist of calculating the minimum or maximum of a set of values. This allows to reuse these operations across different machines without duplicating code.
- Measurements and sensor readings operations should be handled in a separate machine. This machine will be responsible for managing the data from the sensors and providing the necessary

This document is the property of CLEARSY.

operations to access this data. In our example, this abstract machine allow us to test our code without needing to implement the actual sensors.

In B, sets are also considered constants, so they must not be mixed with variables. Consequently, sets shall be declared in a context machine.

There are two ways of accessing assets defined in other machines:

- `SEES` clause: allows a machine to access constants, sets, and operations defined in another machine. However, it does not establish a direct structural dependency between machines. The accessed assets are strictly read-only-they can be used but not modified. Consequently, an operation from a seen machine cannot be invoked if it modifies any of its variables. A single machine can be seen by multiple other machines simultaneously, and such a machine is often referred to as a context machine.
- `IMPORTS` clause: **only permitted in an implementation** and used to import variables and operations from another machine. Unlike `SEES`, it creates a direct structural link, allowing the imported elements to be both accessed and modified within the current machine. To comply with the principle of single implementation, a given machine can be imported by only one implementation.

4.4 Formal specification

To properly structure the code according to **modular architecture** principles, you must create **five distinct components**, each with its own implementation where necessary.

1. **Context machine**: This machine will contain all the constants and sets used in the fuel level alarm system. It must be implemented.
2. **Utils machine**: This machine will contain utility operations, such as calculating the minimum and maximum values. It must be implemented.
3. **Measurements machine**: This machine will simulate the behavior of the sensors, providing operations to read the fuel level and consumption. As we do not plan on implementing actual sensors, this machine will not have an implementation.
4. **Main fuel machine**: This machine will model the fuel level alarm system. It must be implemented.
5. **Main machine**: This machine will be the link between the `Main fuel machine` and the `Context machine`. It must be implemented.

This document is the property of CLEARSY.

A helpful way to visualize the organization of a project is to switch from the default *Classical view* to the *Top-down Graphical view*. This view provides a clear and hierarchical representation of how the different modules are structured and interconnected.

Once all the machines have been created, their specifications defined, necessary implementations completed, and the code fully validated (i.e., all proof obligations discharged), the *Top-down Graphical view* should resemble the following:

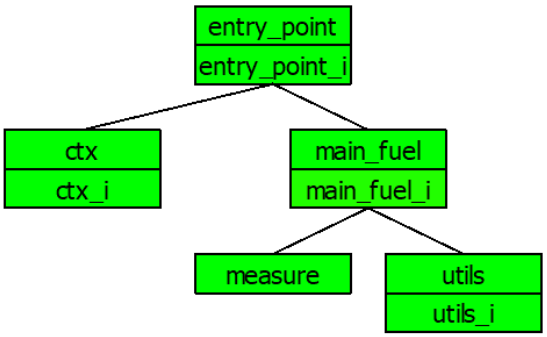


Figure 21: Expected Top-bottom Graphical view on fuel level exercise

In this graphical representation the import dependencies between the various modules are shown. It is important to note that the SEES clause is not represented in this diagram, as it does not establish a direct structural link between machines-it allows read-only access to constants, operations, variables, etc., defined in other machines.

4.4.1 Constants

It is a good practice to declare constants in a so-called context machine. In our case, this machine will contain all the constants used in the fuel level alarm system.

The constants needed in our context are:

- `TANK_CAPACITY`: of type NAT1, the set on natural numbers excluding `0`. The maximum capacity of the tank, which is 1000 liters.
- `MAX_CONSUMPTION`: of type NAT1. The maximum fuel consumption per cycle, which is 10 liters. This variable must be lower than the tank capacity.
- `WARNING_CAPACITY`: of type NAT1. The threshold below which the alarm must be triggered, which is 300 liters. This variable also must be lower than the tank capacity.

This document is the property of CLEARSY.

- ALARM_STATUS: a set of possible alarm states, NOMINAL and LOW_LEVEL. The first state indicates that the fuel level is above the threshold, while the second one indicates that the fuel level is below or equal to the threshold.

Create a ctx machine using both SETS and CONCRETE_CONSTANTS clause to define the constants. Do not forget to precise the **type of each constant**, as well as the **properties they must respect**, inside a PROPERTIES clause.

In order to attribute a value to each constant, the ctx machine must be implemented. The implementation contains a single VALUES clause. Remember that since it is an implementation, each line of VALUES must be separated by a semicolon (;) instead of a double vertical bar (||), which is used in abstract machines.

4.4.2 Utils

This machine will contain utility operations, which use no variables besides those passed as parameters, that can be reused across different machines. In our case, we will define two operations: minimum and maximum. The first one calculates the minimum value between two entries, the two ultrasonic level sensors. The second one calculates the maximum value between three entries, the three flowmeters.

You may simply create an abstract machine named utils, which can use the abstract operations max and min. Here you have the specification for the minimum operation:

```
rr <-- minimum(aa, bb) =
PRE
   aa: NAT &
   bb: NAT
THEN
   rr := min({aa, bb})
END
```

Since min do not belong to B0 language, in its implementation, we may choose a conditional approach:

```
rr <-- minimum ( aa , bb) =
BEGIN
   IF aa >= bb
   THEN
       rr := bb
   ELSE
       rr := aa
```


This document is the property of CLEARSY.

```
   END
END
```

The next step is to insert the specification and implementation of the `minimum` operation into the `OPERATIONS` clause of the abstract machine and its corresponding implementation.

Following the same approach, a `maximum` operation should then be defined to calculate the maximum value among three inputs. This should include both its specification in the abstract machine and its implementation.

4.4.3 Measurements

In this exercise, handling real sensor readings is not required. Instead, create an abstract machine named `measure` that simulates the behavior of the sensors.

Besides a `SEES` class, in order gain read-only access over the `ctx` constants, two operations will be defined in this machine:

- `measure_level`: simulates the reading of the two ultrasonic level sensors and returns their values. This operation takes no parameters and returns two natural numbers.
- `measure_consumption`: simulates the reading of the three flowmeters and returns their values. This operation takes no parameters and returns three natural numbers.

To achieve this, you can use the `::` abstract operator to indicate that a variable takes an arbitrary value from a set, without specifying exactly which one. For example:

```
m1 :: 0..TANK_CAPACITY
```

Inside the `measure` abstract machine, do the specification for both operations. There is no need for implementation since we do not plan on implementing actual sensors. Consider this machine as a mockup, which will be used only to test the code. In order to properly use its operations, they must, later on, manually be coded in C.

4.4.4 Main fuel machine

Finally, create an abstract machine named `main_fuel` modeling the fuel level alarm system. This machine will use the constants defined in the `ctx` machine through a `SEES` clause.

We need three variables in the `VARIABLES` clause to represent the fuel level alarm system:

This document is the property of CLEARSY.

- `estimated_level`: the estimated amount of fuel in the tank, which is an integer between 0 and the tank capacity. It should be initialized to `0`.
- `estimated_consumption`: the estimated amount of fuel consumed, which is an integer between 0 and the maximum consumption per cycle. It also should be initialized to `0`.
- `status`: the status of the fuel level alarm, which type is `ALARM_STATUS`. It should be initialized to `LOW_LEVEL` since before any readings we estimate an empty tank.

To ensure the system's correct behavior, we will define an `INVARIANT` that guarantees the following properties:

- The typing of each variable.
- `(status /= LOW_LEVEL => estimated_level > WARNING_CAPACITY)`: This security property ensures that the alarm is triggered whenever the estimated fuel level is below or equal to the threshold.

Note that the previous property is the contrapositive of the implication `(estimated_level <= WARNING_CAPACITY => status = LOW_LEVEL)`. However, the contrapositive expresses a more restrictive behavior, which is desirable in the context of safety-critical systems. In such systems, we prefer to erroneously prevent the aircraft from flying due to a false alarm, rather than risk allowing it to fly with insufficient fuel. Although both invariants generate the same proof obligations, the contrapositive formulation better reflects the safety-first mindset required in such scenarios.

Create the `SEES`, `VARIABLES`, `INVARIANT` and `INTIALISATION` clauses, in this exact order. All needed variables description along with their initial values, as well as the invariant, are detailed above.

This abstract machine, has five `OPERATIONS`, of which 3 getters:

- `compute_initial_level`: initial estimation of the amount of fuel in the tank. It changes `estimated_level` based on a new value belonging to the range `0..TANK_CAPACITY` and `status` which takes a value on the `ALARM_STATUS`. If `status` is different from `LOW_LEVEL`, then the `estimated_level` must be greater than the `WARNING_CAPACITY`.

```
compute_initial_level =
   BEGIN
       estimated_level, status :(
           estimated_level: 0..TANK_CAPACITY &
           status : ALARM_STATUS &
           (status /= LOW_LEVEL => estimated_level > WARNING_CAPACITY))
   END
```


This document is the property of CLEARSY.

- `compute_remaining_fuel`: called repeatedly to estimate consumption and remaining fuel. It changes `estimated_level` based on a new value belonging to the range `0..TANK_CAPACITY`, `estimated_consumption` which takes a new value in the range `0..MAX_CONSUMPTION` and `status` which takes a value on the `ALARM_STATUS`. The new value of `estimated_level` must be lower or equal to the old one since the aircraft consumes fuel. If `status` is different from `LOW_LEVEL`, then the `estimated_level` must be greater than the `WARNING_CAPACITY`.

```
compute_remaining_fuel =
BEGIN
    estimated_level, estimated_consumption, status :(
        estimated_level: 0..TANK_CAPACITY &
        estimated_consumption : 0..MAX_CONSUMPTION &
        status : ALARM_STATUS &
        (estimated_level <= estimated_level$0) &
        (status /= LOW_LEVEL => estimated_level > WARNING_CAPACITY))
END
```

- `get_estimated_level`, `get_status` and `get_estimated_consumption`: these operations return the current values of `estimated_level`, `status`, and `estimated_consumption`, respectively. Here you have the specification for `get_estimated_level`:

```
estimated_lvl <-- get_estimated_level =
   BEGIN
       estimated_lvl := estimated_level
   END;
```

Create the specifications for the other two getters in a similar way, following the same pattern.

4.4.5 Entry-point machine

Since `SEES` does not provide a direct structural link between machines, you must create an entry-point machine that imports both `main_fuel` and `ctx`. The `entry_point` machine is the entry point of the system.

Inside the `entry_point` machine specification, all `main_fuel` operations are redefined in the `OPERATIONS` clause, allowing them to be called from the main program. The redefined operations specification only stats that the return variable takes a value from its type, without specifying the exact one. This is because the entry-point machine does not implement the operations, it only provides a way to call them. For example:

This document is the property of CLEARSY.

```
status_1 <-- get_status_entry_point =
    BEGIN
        status_1 :: ALARM_STATUS
    END;
```

For the operations not returning anything, `compute_initial_level` and `compute_remaining_fuel`, they may be defined as `skip`. Here you have the specification for the `entry_point` machine:

```
MACHINE
    entry_point

SETS
   ALARM_STATUS = {NOMINAL, LOW_LEVEL}

OPERATIONS
    compute_initial_level_entry_point = skip;

    compute_remaining_fuel_entry_point = skip;

    estimated_lvl <-- get_estimated_level_entry_point =
    BEGIN
        estimated_lvl :: NAT
    END;

    status_1 <-- get_status_entry_point =
    BEGIN
        status_1 :: ALARM_STATUS
    END;

    estimated_cmpt <-- get_estimated_consumption_entry_point =
    BEGIN
        estimated_cmpt :: NAT
    END
END
```

Note that we redefined the `ALARM_STATUS` set. Since it is defined inside `ctx` and the goal of `entry_point` is to link both `ctx` and `main_fuel`, it is strictly forbidden to use the `SEES` clause. Therefore, `IMPORTS` are also forbidden in the specification, so `ALARM_STATUS` must be redefined in order to properly build a specification for `get_status_entry_point`.

The implementation of the `entry_point` machine is straightforward. It uses the `IMPORTS` clause to import the `main_fuel` and `ctx` machines, allowing it to access their operations and variables, creating

This document is the property of CLEARSY.

a direct link between them. Each entry-point's operation may call its equivalent on `main_fuel`, and return the same value:

```
IMPLEMENTATION entry_point_i
REFINES entry_point

IMPORTS ctx, main_fuel

OPERATIONS
   compute_initial_level_entry_point =
   BEGIN
       compute_initial_level
   END;

   compute_remaining_fuel_entry_point =
   BEGIN
       compute_remaining_fuel
   END;

   estimated_lvl <-- get_estimated_level_entry_point =
   BEGIN
       estimated_lvl <-- get_estimated_level
   END;

   status_1 <-- get_status_entry_point =
   BEGIN
       status_1 <-- get_status
   END;

   estimated_cmpt <-- get_estimated_consumption_entry_point =
   BEGIN
       estimated_cmpt <-- get_estimated_consumption
   END

END
```

4.5 Implementation

Now that the abstract `main_fuel` machine is defined, its implementation must be built.

It will use the utility operations from the `utils` machine and the sensor readings from the `measure` machine to compute the initial fuel level alongside the remaining fuel level, both machines must be

This document is the property of CLEARSY.

imported via the IMPORTS clause. Do not forget to see ctx to access the constants defined there.

```
SEES ctx
IMPORTS
    measure,
    utils
```

All variables declared in the specification must be declared in the CONCRETE_VARIABLES clause of the implementation. The now concrete variables must have the same initial previous values defined on an INITIALISATION clause.

There is no need to redefine the INVARIANT as it is automatically inherited from the abstract machine.

With the purpose of implementing the OPERATIONS. We will use the following approach for compute_initial_level:

1. We declare two local variables, m1 and m2, to store the readings from the two ultrasonic level sensors.
2. We call the measure_level operation from the measure machine to get the readings from the sensors, which will be stored in m1 and m2.
3. We use the minimum operation from the utils machine to compute the initial fuel level as the minimum of the two sensor readings.
4. If estimated_level <= WARNING_CAPACITY, we must set the status to LOW_LEVEL, otherwise we set it to NOMINAL.

```
compute_initial_level =
   VAR m1, m2 IN
       m1, m2 <-- measure_level;
       estimated_level <-- minimum(m1, m2);
       IF estimated_level <= WARNING_CAPACITY
       THEN
           status := LOW_LEVEL
       ELSE
           status := NOMINAL
       END
   END
   ;
```

Besides, we will use the following approach for compute_remaining_fuel:

This document is the property of CLEARSY.

1. We declare three local variables, `m1`, `m2`, and `m3`, to store the readings from the three flowmeters.
2. We call the `measure_consumption` operation from the `measure` machine to get the readings from the flowmeters, which will be stored in `m1`, `m2`, and `m3`.
3. We use the `maximum` operation from the `utils` machine to compute the fuel consumption as the maximum of the three flowmeter readings.
4. We update the `estimated_level` by subtracting the `estimated_consumption` from the previous value. The fuel level cannot be negative, so we ensure that the new estimated level is not less than 0.
5. If the new `estimated_level` is less than or equal to the `WARNING_CAPACITY`, we set the `status` to `LOW_LEVEL`, otherwise we set it to `NOMINAL`.

```
compute_remaining_fuel =
   VAR m1, m2, m3 IN
       m1, m2, m3 <-- measure_consumption;
       estimated_consumption <-- maximum(m1,m2,m3);
       IF estimated_consumption >= estimated_level
       THEN
           estimated_level := 0
       ELSE
           estimated_level := estimated_level - estimated_consumption
       END;
       IF estimated_level <= WARNING_CAPACITY
       THEN
           status := LOW_LEVEL
       END
   END
```

The three getter operations `get_estimated_level`, `get_status`, and `get_estimated_consumption` are straightforward. They simply return the current values of the corresponding variables, for example:

```
estimated_lvl <-- get_estimated_level =
   BEGIN
       estimated_lvl := estimated_level
   END;
```

After completing the implementation, you will have a complete modeling of the fuel level alarm system. The implementation refines the abstract machine using the utility operations and sensor

readings to compute the initial fuel level and the remaining fuel. This assures proper **modular architecture**, allowing for better organization and maintainability of the code.

4.6 Proving the implementation

After executing  for the given model, for each machine, the output should be as follows:

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
ctx	OK	OK	0	0	0	-
ctx_i	OK	OK	5	5	0	-
entry_point	OK	OK	0	0	0	-
entry_point_i	OK	OK	2	2	0	-
main_fuel	OK	OK	2	2	0	-
main_fuel_i	OK	OK	34	33	1	-
measure	OK	OK	0	0	0	-
utils	OK	OK	4	4	0	-
utils_i	OK	OK	6	6	0	-

Figure 22: Expected Atelier B main screen after running the automatic prover

As you can see, there is one Proof obligation left for the `main_fuel_i` machine. There are multiple ways to discharge this proof obligation, but the most straightforward is to use a stronger, but slower, level of automatic prover, such as  or  . They can be used after **right-clicking the component** > **Proof** > **Automatic (Force 2)**. This is the expected final result:

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
ctx	OK	OK	0	0	0	-
ctx_i	OK	OK	5	5	0	-
entry_point	OK	OK	0	0	0	-
entry_point_i	OK	OK	2	2	0	-
main_fuel	OK	OK	2	2	0	-
main_fuel_i	OK	OK	34	34	0	-
measure	OK	OK	0	0	0	-
utils	OK	OK	4	4	0	-
utils_i	OK	OK	6	6	0	-

Figure 23: Expected Atelier B main screen after running the automatic prover

This document is the property of CLEARSY.

Stronger levels of automatic prover may take longer to run, especially for larger machines, and they do not guarantee to succeed the discharge. Their complexity is exponentially higher than the weaker levels, so they may never complete. If you are working on a large machine, it is recommended to manually discharge the Proof Obligations via the interactive prover instead.

4.7 Generating and testing the C code

Before generating the C code, make sure each implementation model passes the B0 check. To do this, select each implementation component in the main view and use the **Component** > **B0 Check** menu or press `Ctrl+B`. This verification must be performed for each of the following implementations: `ctx_i`, `utils_i`, `main_fuel_i`, and `entry_point_i`.

Some abstract machines, such as `measure`, do not have a B0 implementation and therefore cannot be automatically translated into C. However, the code generator can still produce the corresponding header files. To do this, select the `measure` machine in the main view and use the **Component** > **Code generator** menu option. This will generate only the C header file `measure.h`, which contains the declarations of the operations defined in the `measure` machine.

For abstract machines without an implementation (in this case, only `measure`), you must manually create and write the associated C source file to provide the required functionality. Below is an example of C code you can create and write manually to simulate the behavior of the `measure` machine during testing and execution:

```c
/* File: measure_bs.c */
#include "measure.h"
#include <stdlib.h>


void measure__INITIALISATION() {

}

void measure__measure_level(int32_t *m1, int32_t *m2) {
    *m1 = rand() % ctx__TANK_CAPACITY;
    *m2 = rand() % ctx__TANK_CAPACITY;
}

void measure__measure_consumption(int32_t *m1, int32_t *m2, int32_t *m3) {
    *m1 = rand() % ctx__MAX_CONSUMPTION;
    *m2 = rand() % ctx__MAX_CONSUMPTION;
```


This document is the property of CLEARSY.

```
    *m3 = rand() % ctx__MAX_CONSUMPTION;
}
```

To test your code, add the following `main.c` file to your project:

```c
/* File: main.c */
#include "entry_point.h"
#include <stdio.h>
#include <stdlib.h>


int main() {
    // Seed the random number generator
    srand(55);

    entry_point__INITIALISATION();
    entry_point__compute_initial_level_entry_point();

    int32_t estimated_lvl;
    entry_point__get_estimated_level_entry_point(&estimated_lvl);
    printf("The initial fuel level is: %d\n", estimated_lvl);

    for (int i = 0; i < 10; i++) {
        entry_point__compute_remaining_fuel_entry_point();
        entry_point__get_estimated_level_entry_point(&estimated_lvl);

        int32_t estimated_consumption;
        entry_point__get_estimated_consumption_entry_point(&estimated_consumption);

        ctx__ALARM_STATUS status;
        entry_point__get_status_entry_point(&status);

        printf("The estimated fuel level after cycle %d is: %d \n", i + 1,
↪   estimated_lvl);
        printf("With a estimated consumption of: %d \n", estimated_consumption);
        printf("The alarm status is: %s \n", (status == 1 ? "LOW_LEVEL" : "NOMINAL"));
        printf("=========================================\n");
    }
}
```

It initializes the entry_point machine, computes the initial fuel level, and then simulates 10 cycles

This document is the property of CLEARSY.

of fuel consumption. In each cycle, it retrieves the estimated fuel level, estimated consumption, and alarm status, printing them to the console.

Create a new Makefile, similar to the one provided in the previous exercises, to compile and run your code. Once the code compiled and successfully executed, the output may be similar to the following:

```
The initial fuel level is: 325
The estimated fuel level after cycle 1 is: 323
With a estimated consumption of: 2
The alarm status is: NOMINAL
========================================
The estimated fuel level after cycle 2 is: 319
With a estimated consumption of: 4
The alarm status is: NOMINAL
========================================
The estimated fuel level after cycle 3 is: 315
With a estimated consumption of: 4
The alarm status is: NOMINAL
========================================
The estimated fuel level after cycle 4 is: 306
With a estimated consumption of: 9
The alarm status is: NOMINAL
========================================
The estimated fuel level after cycle 5 is: 299
With a estimated consumption of: 7
The alarm status is: LOW_LEVEL
========================================
The estimated fuel level after cycle 6 is: 290
With a estimated consumption of: 9
The alarm status is: LOW_LEVEL
========================================
The estimated fuel level after cycle 7 is: 282
With a estimated consumption of: 8
The alarm status is: LOW_LEVEL
========================================
The estimated fuel level after cycle 8 is: 273
With a estimated consumption of: 9
The alarm status is: LOW_LEVEL
========================================
The estimated fuel level after cycle 9 is: 264
With a estimated consumption of: 9
The alarm status is: LOW_LEVEL
```

```
========================================
The estimated fuel level after cycle 10 is: 256
With a estimated consumption of: 8
The alarm status is: LOW_LEVEL
========================================
```

4.8 To go further

- **Flight time constraints**: Add a new variable to `main_fuel` which tracks for how long the aircraft has been flying. The variable should be incremented by 1 every time the `compute_remaining_fuel` operation is called. If the aircraft has been flying for more than 100 cycles, the alarm should be triggered regardless of the fuel level. This will ensure that the pilot is alerted to potential fuel shortages even if the fuel level is above the threshold, as long as the aircraft has been flying for an extended period.
- **Time-based alarm constraints**: Similar to the previous one, add a new variable to `main_fuel` which tracks the time since the last alarm has been triggered. If it has been triggered for more than 10 cycles, the status must go from `LOW_LEVEL` to **EMERGENCY mode**.

This document is the property of CLEARSY.

5 A simple loop usage example

5.1 Introduction

This exercise introduces the concept of **loops** in the B Method. Loops are a fundamental construct in programming and can be particularly challenging to verify due to the need for properly defined **invariants** and **variants**. Which are essential for proving the correctness of the loop and its termination in a finite number of iterations.

To explore this concept, we will examine a simple loop whose goal is to copy the value of `x` into `y`, and then increment `x` by 1, three times. This operation is logically equivalent to `x := y + 3`.

Through this exercise, you will:

- **Prove termination**: Ensure the loop will eventually terminate by defining a proper **variant**, a positive and decreasing expression.
- **Maintain invariants**: Identify an **invariant**, a condition that must hold true throughout the loop's execution, to ensure the loop behaves as expected.

Finding suitable invariants and variants is often not trivial. It's common to discover that a loop may be incorrect, requiring a redesign. However, both are needed for the B Method to prove the loop is correct and terminates, respectively. Even experienced B developers frequently revisit their loop construction to satisfy proof obligations.

Always keep in mind: a loop can meet its specification but still fail the proof if its invariant or variant is incorrect.

5.2 Natural language description

The goal of this exercise is to model a simple loop-based operation that maintains the relationship: `xx = yy + 3`

Start with two integer variables, `xx` and `yy`. The loop should run exactly three times. In each iteration:

- The current value of `yy` is copied into `xx`.
- The value of `xx` is incremented by 1.

This guarantees that after three iterations, `xx` will be exactly `yy` + 3.

This document is the property of CLEARSY.

However, in this version of the model, we take a more functional and user-friendly approach: Rather than running a loop explicitly, we allow the user to set a new value for `yy`, and the system will automatically update `xx` accordingly, preserving the invariant `xx = yy + 3`.

In B, single characters are not allowed as variable/constant names. They are used in special use cases as Proof Jokers.

5.3 Formal specification

Firstly, create a new machine. Our goal is clear: declare two constants, `xx` and `yy`, both initially respecting the invariant `xx = yy + 3`. Then create an operation `calculate_new_x` taking a `new_yy` as parameter. Both values of `yy` and `xx` will be updated to `new_yy` and `xx` incremented by 3, thus obtaining `yy := new_yy` and `xx := new_yy + 3` which respects the invariant `xx = yy + 3`.

Later on the implementation, we will look for incrementing `xx` by 1 three times, obtaining the same result but using a loop instead.

We define the abstract machine containing the desired specification:

```
MACHINE
    Calculate_X_from_Y

CONCRETE_VARIABLES
    xx, yy

INVARIANT
    xx : INT &
    yy : INT &
    yy < (MAXINT - 3) &
    xx = yy + 3

INITIALISATION
    xx := 3 ||
    yy := 0

OPERATIONS
   calculate_new_x (new_yy) =
   PRE
       new_yy : INT &
       new_yy < (MAXINT - 3)
   THEN
```


This document is the property of CLEARSY.

```
      yy := new_yy ||
      xx := new_yy + 3
   END
END
```

This is a pretty standard machine compared to the ones we have seen so far. We may dissect it to understand its components:

- It contains the `CONCRETE_VARIABLES` clause, which declares the concrete variables `xx` and `yy`, both of type `INT`.
- The `INVARIANT` clause states that `xx` and `yy` are integers and that `xx` must always equal `yy + 3`. We also must ensure that `yy` is less than `MAXINT - 3` to avoid overflow when we increment `xx` by 3.
- The `INITIALISATION` clause sets the initial values of `xx` to 3 and `yy` to 0.
- The `OPERATION` clause defines the operation `calculate_new_x` taking `new_yy` as parameter, which sets `yy := new_yy` and `xx := new_yy + 3`.

The example may seem trivial, but it is a good starting point to understand how loops work in B.

5.4 Implementation

Besides, we define the implementation model, which will implement a loop to achieve the same result, `xx := yy + 3`:

```
IMPLEMENTATION Calculate_X_from_Y_i
REFINES Calculate_X_from_Y

INITIALISATION
   xx := 3;
   yy := 0

OPERATIONS
   calculate_new_x (new_yy) =
   VAR index, xx_copy IN
      // index: loop's control index
      // xx_copy: local copy of xx that is progressivelly incremented by 1
      index := 0;
      xx_copy := new_yy;
```


This document is the property of CLEARSY.

```
      WHILE index < 3 DO
         xx_copy := xx_copy + 1;
         index := index + 1
      INVARIANT
         xx_copy : INT &
         index : NAT &
         index: 0..3 &
         xx_copy = new_yy + index
      VARIANT
         3 - index
      END;
      xx := xx_copy;
      yy := new_yy
   END
END
```

Similarly to C or ADA, a loop is always preceded by an initialization phase where all nested loop variables are initialized.

For the purpose of giving a hand to the automatic prover, we may define a `xx_copy` variable. It will be used to store the local value of `xx` at each iteration, therefore `xx_copy` can be updated multiple times inside the loop. This variable is initialized to `new_yy`. At the very end, `xx_copy` will be equal to `new_yy + 3`, and we can simply copy its value to `xx` via `xx := xx_copy` alongside coping `new_yy` to `yy` via `yy := new_yy`.

The invariant must assure these properties:

1. At the beginning of the loop, after the initialization phase, the invariant must be true. In our example, with `index := 0` and `xx_copy := new_yy`, the invariant is true.
2. If true at the beginning and if the loop continuation condition is respected, the iterations must conserve its invariant. In our example, if `index < 3`, `index : 0..3` and `xx_copy := new_yy + index`, then the invariant is still true if we replace `index` by `index + 1` and `xx_copy` by `xx_copy + 1`.
3. At the very end, if the invariant is preserved and the loop's termination condition is met, we must deduct the property derived from the specification. In our example, when `index >= 3`, we can conclude that `xx_copy = new_yy + 3`.

The first property must be always true in the machine's context, therefore the other two must be true for all variables verifying the invariant.

With the intention of assuring these properties, here are some examples of what may be included in the invariant:

This document is the property of CLEARSY.

- The type of the loop variables: `index : 0..3`, assuring that the variant is never negative, and `xx_copy : NAT`. In our example, `xx` and `yy` are already typed on the `PROPERTIES` clause.
- The properties to be respected after each iteration, `xx_copy = new_yy + index` in our example.

5.5 Proving the implementation

As you are used to, you can now run  . Everything should be proved automatically. You can then run the B0 check on the component to ensure it is B0 compliant, which means it can be translated to C code.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
Calculate_X_from_Y	OK	OK	5	5	0	OK
Calculate_X_from_Y_i	OK	OK	10	10	0	OK

Figure 24: Expected Atelier B main screen after running the automatic prover

5.6 Generating and testing the C code

In the main view, navigate to **Component** > **B0 Check** to ensure that the implementation is valid, consistent with the abstract specification, and ready for code generation.

Once the implementation is B0 checked, generate the corresponding proved C code using the built-in translator.

In order to run and test the code, add this main function to the project:

```c
/* File: main.c */
#include <stdio.h>
#include "Calculate_X_from_Y.h"

int main(void)
{
    Calculate_X_from_Y__INITIALISATION();
    printf("Initialization: xx = %d and yy = %d\n", Calculate_X_from_Y__xx,
↪   Calculate_X_from_Y__yy);
    const int new_yy = 12;
    printf("Calculate new value of xx when yy = %d\n", new_yy);
    Calculate_X_from_Y__calculate_new_x(new_yy);
```


This document is the property of CLEARSY.

```
    printf("After calculate new X: xx = %d yy = %d\n", Calculate_X_from_Y__xx,
↪   Calculate_X_from_Y__yy);
}
```

This `main.c` file outputs the initial values of `xx` and `yy`, then calculates their new values based on the `new_yy` variable. Compile it using your makefile and run your code.

Here is a trace of the expected result:

```
Initialization: xx = 3 and yy = 0
Calculate new value of xx when yy = 12
After calculate new X: xx = 15 yy = 12
```

5.7 To go further

- **Offset value of x passed as a parameter**: Change the loop so that the number of iterations depends on a `new_offset` parameter, which is passed to the `calculate_new_x` operation along with `new_yy`. The `new_offset` value should update an `ABSTRACT_VARIABLE` called `offset`, which will be used to define the invariant correctly. At the end of the loop, the result should be `xx := new_yy + new_offset`. You may need to update both the machine's and the loop's invariants to reflect these changes.

This document is the property of CLEARSY.

6 Filling an array with a given value

6.1 Introduction

The following exercise is another example of loop usage in B, this time, with a more real life example since we will be iterating trough arrays! The goal is clear: there will be an array filled with zeros, write an operation that fills it with its identity, using a loop.

At the end of this exercise, you should:

- **Better understand variants and invariants**: how to find a suited invariant? How to assure the loop terminates?
- **Understand B arrays**: Similar to arrays in other programming languages, B arrays are a collection of elements indexed by indexes, but with a slightly more mathematical approach.
- **Discover abstract iterators**: A very smart way to iterate over arrays in B, which will be used in the following exercises.

This exercise deepens your understanding of loop structures in B by exploring array manipulation, while introducing key concepts like invariants, variants, and abstract iterators in a more practical, real-world context.

6.2 Natural language description

Given an **array** of size 11, from 0 to 10, initialized with zeros, the goal is to write an operation that fills the array with its identity. The value on the index `0` should be set to `0`, index `1` to `1`, and so on, until index `10` which should be set to `10`. The operation should iterate through each index of the array, setting its value to its identity. The iterations must be done using an **abstract iterator**.

6.3 Arrays

Since we are talking about loops, we must have something to loop over. That is why we will introduce the concept of **arrays**.

6.3.1 Declaring arrays

An array in the B is typically represented as a total function from a finite index domain (such as `0..NN`) to a set of values. An array of size `NN + 1` with elements of type `TT` is represented as:

```
array : 0..NN --> TT
```

Note that "- ->" represents a total function

As an example, let NN $= 2$ and TT $\subseteq$ `INT`. In this case, arrays can be initialized as follows:

```
array1 := {0 |-> 0, 1 |-> 0, 2 |-> 0}

...

array2 := (0..NN) * {0}
```

Note that "|->" represents the mapping from index to value (x, y). Also, "*" represents the Cartesian product, the set of pairs x |-> y such that x : (0..NN) and y : {0}.

6.3.2 Domain and image

The **domain** of an array is the set of indices over which the array is defined. For example, for `array1` defined as above, `dom(array1) = {0, 1, 2}`.

The **image** of an array is the set of values that the array can take. For `array1`, `ran(array1) = {0}` since all elements are initialized to 0.

Both **dom** and **ran** are considered abstract operations, therefore cannot be used in the implementation elsewhere than the abstract level, such as in the invariant/variant or properties of the machine.

However, they can and should be used in the specification to define properties of the array. We must transform them into concrete operations through loops.

6.4 Abstract iterator

6.4.1 Abstract machine

Now that we properly understand B arrays, we need a way of iterating over them. With this goal, an **abstract iterator** may be the best option for B enthusiasts like you and me. An abstract iterator is an easy way of iterating over elements of an array using two sets, `Done` and `Todo`.

This document is the property of CLEARSY.

The **abstract iterator** is very useful for eliminating the need to calculate the array's next element and for avoiding arithmetic operations.

Here you have its abstract machine:

```
MACHINE
    iter_services

SEES
    ctx

ABSTRACT_VARIABLES
    Todo,
    Done

INVARIANT
    Todo <: (0..NN) &
    Done <: (0..NN) &
    Todo \/ Done = (0..NN) &
    Todo /\ Done = {}

INITIALISATION

    Todo := (0..NN) ||
    Done := {}

OPERATIONS
    flag <-- init_iter =
    BEGIN
        Todo := (0..NN) ||
        Done := {} ||
        flag := bool ((0..NN) /= {})
    END
    ;

    flag, elt <-- next_iter =
    PRE
        Todo /= {}
    THEN
        ANY
            chosen,
            new_Todo
        WHERE
```


This document is the property of CLEARSY.

```
            chosen : NAT &
            chosen : Todo &
            new_Todo = Todo - {chosen}
        THEN
            Todo := new_Todo ||
            Done := Done \/ {chosen} ||
            flag := bool (new_Todo /= {}) ||
            elt := chosen
        END
    END

END
```

At the beginning, `Todo` contains all indices of the array, while `Done` is empty. The `init_iter` is mostly used to set the value of `flag`, its returned variable, which is always `True` since `NN >= 0`, `NN : NAT`.

The `next_iter` operation retrieves the next index from `Todo`, adds it to `Done`, and updates `flag` to indicate whether there are more elements to process. This operation is designed to keep being used until `flag` becomes `False`, meaning there are no more elements to iterate over.

A common mistake is to use `flag := bool(Todo /= {})`, in the `init_iter` operation, instead of `flag := bool((0..Array_size) /= {})`. Even if `Todo := (0..NN)`, it is strictly forbidden to use it in another substitution inside `init_iter`, since they are all done in parallel. The `Todo` variable is not yet defined at this stage, so it cannot be used to set the value of `flag`. Instead, we must use the `NN` constant, which is defined in the `ctx` machine.

6.4.2 Constants

You may note the absence of `NN` declaration as a constant. As explained on the modular architecture section, constants and variables should be declared in the context machine.

Create a new machine called `ctx` and declare the `NN` constant inside its `CONCRETE_CONSTANTS` clause. Copy and paste the following code into your `ctx` machine:

```
MACHINE
    ctx

CONCRETE_CONSTANTS
    NN
```


This document is the property of CLEARSY.

```
PROPERTIES
    NN : NAT

END
```

This machine must be implemented in order to properly give a value to the NN constants. Build its implementation which will only contain a VALUES clause defining the value of NN:

```
IMPLEMENTATION ctx_i
REFINES ctx

VALUES
    NN = 10

END
```

6.4.3 Implementation of the abstract iterator

After constructing the abstract iterator, it must be implemented. This machine will refine the abstract iterator and provide a concrete implementation of the next_iter operation.

```
IMPLEMENTATION iter_services_i
REFINES iter_services

SEES
    ctx

CONCRETE_VARIABLES

    index

INVARIANT

    index : NAT &
    Todo = index..NN &
    Done = 0..(index-1)

INITIALISATION

    index := 0
```


This document is the property of CLEARSY.

```
OPERATIONS

    flag <-- init_iter =
    BEGIN
        index := 0 ;
        flag := bool (index <= NN)
    END
    ;

    flag, elt <-- next_iter =
    BEGIN
        elt := index;
        index := index + 1 ;
        flag := bool (index <= NN)
    END

END
```

The implementation version of the `next_iter` simply returns the current index incremented by 1, accompanied by the `flag` variable which indicates whether we are still on the array's scope.

Abstract iterators are an insightful technique to abstract the complexity of iterating over arrays, allowing you to focus on the logic of your operations without worrying about the underlying mechanics of iteration. You simply need to import `iter_services` to have access to the abstract iterator operations:

```
IMPORTS
    iter_services
```

And then use the `init_iter` and `next_iter` operations to iterate over the array on the implementation phase. Imported operations can be used following this syntax:

```
    flag <-- init_iter ;
    ...
    flag, index <-- next_iter ;
```

The provided code for the abstract iterator must be considered as a base-layer open source code, it can be either used as it is or modified to fit your needs.

This document is the property of CLEARSY.

6.5 Formal specification

You have now a powerful B development toolkit: Loops, arrays and abstract iterators. Proceed to the next step: implementing an operation which fills an array with a given value. Watch out, with great power comes great responsibility.

You may consider this approach:

1. SEES the `ctx` machine in order to gain access over `NN`.
2. Create an `array` variable, which will be used to store the values. It must be a total function mapping indices from `0` to `NN` to natural numbers `NAT`, all zero at the very beginning.

```
SEES
    ctx

CONCRETE_VARIABLES
    Array

INVARIANT
    Array : (0..NN) --> NAT

INITIALISATION
    Array := (0..NN) * {0}
```

3. Define an operation called `set_array_value` with no parameters. The operation must iterate through all array indexes and set each value to its corresponding index (its identity).

Start by creating a new machine, which will contain the abstract specification of the array filling operation. Copy and paste the following code:

```
MACHINE
    array

SEES
    ctx

CONCRETE_VARIABLES
    Array

INVARIANT
    Array : (0..NN) --> NAT
```


This document is the property of CLEARSY.

```
INITIALISATION
    Array := (0..NN) * {0}

OPERATIONS
    set_array_value =
    BEGIN
        Array := id(0..NN)
    END

END
```

Let's take a closer look at the `set_array_value` operation: The identity function `id` returns a relation from a set: the relation that associates each element of the set with itself.

```
id({2, 3, 4}) = {2 |-> 2, 3 |-> 3, 4 |-> 4}
```

6.6 Implementation

In B, just like in most programming languages, when we want to go through all elements of an array, we use a loop. So how do we build this loop? We have seen from the previous example that we must find a suited invariant/variant pair.

Starting from the variant

1. To iterate from the beginning to the end of the array, since the array goes from `0` to `NN`, needs exactly `NN + 1` iterations.
2. To control the loop, introduce a `flag` variable, which keeps being true while there are elements to be treated. Initially set as `True`, since the abstract iteration does not support empty arrays (`PRE Todo /= {} THEN`).
3. Using the abstract iterator, each interation will remove an element from the `Todo` set and add it to the `Done` set. The loop will continue until there are no more elements in the `Todo` set.
4. A natural choice for the variant may be `card (Todo)`, the number of elements present on `Todo`, used as our control loop index. When it reaches `0`, the loop will terminate, since there are no more elements to iterate over.
5. The newfound variant is a positive expression that decreases with each iteration, ensuring the loop will eventually terminate.

This document is the property of CLEARSY.

```
VARIANT
    card (Todo)
```

Followed by the invariant

Invariants are quite a bit more challenging to find, compared to variants. As we have seen in the previous exercise, the invariant must be true at the beginning of the loop, preserved during each iteration, and true at the end of the loop.

In addition, the invariant must be strong enough to help the automatic prover as much as possible. Very often, even with the best suited invariant, it is impossible to fully prove everything without interactive proof. Since interactive proof is not the goal of this exercise, it will be treated on following exercises, you may use this well suited invariant:

```
INVARIANT
    Todo \/ Done = (0..NN) &
    flag = bool (Todo /= {}) &
    Array : (0..NN) --> NAT &
    (Done /= {} => !(xx).(xx : Done => Array(xx) = xx))
```

- `Todo \/ Done = (0..NN)` ensures all indices of the array are either in `Todo` or `Done`, meaning we have not missed any index.
- `flag = bool (Todo /= {})` ensures the loop continues as long as there are elements to iterate over.
- `Array : (0..NN) --> NAT` ensures the array is a total function mapping indices from `0` to `NN` to natural numbers (NAT).
- `(Done /= {} => !(xx).(xx : Done => Array(xx) = xx))` ensures that if `Done` is not empty, then all already treated indexes now have themselves as value. This is a strong invariant that helps the automatic prover to understand that the array is updated correctly.

You can now implement the operation in a new machine, which will contain the concrete implementation of the `set_array_value` operation. Do not forget to import the abstract iterator and declare a `flag` loop variable, as it will be used to iterate through the array:

```
IMPLEMENTATION array_i
REFINES array

SEES
    ctx
```


This document is the property of CLEARSY.

```
IMPORTS
    iter_services

INITIALISATION
    Array := (0..NN) * {0}

OPERATIONS
    set_array_value =
    VAR
        current, flag
    IN
        flag <-- init_iter ;
        WHILE
            flag = TRUE
        DO
            flag, current <-- next_iter ;
            Array(current) := current
        INVARIANT
            Todo \/ Done = (0..NN) &
            flag = bool (Todo /= {}) &
            Array : (0..NN) --> NAT &
            (Done /= {} => !(xx).(xx : Done => Array(xx) = xx))
        VARIANT
            card (Todo)
        END
    END
END
```

In this implementation: `flag` is the loop condition, `current` is an element of `Done` indicating the next element to treat.

Note that `array_i`, `iter_services_i` and `iter_services` are not shown in green since they lack some POs to be discharged.

As said on the entry-point machine section of the fuel level exercise, since SEES does not provide a direct structural link between machines, you must create an entry-point machine that imports both `array` and `ctx`. It will be entry-point of the system.

It must redefine the `set_array_value` operation as a skip, since it will be simply called without parameters. The main machine specification should look like this:

```
MACHINE
    entry_point

OPERATIONS
    op = skip
END
```

In order to implement, the `entry_point` machine may import both `array` and `ctx`, and its `op` operation shall delegate to `set_array_value` from `array`.

```
IMPLEMENTATION entry_point_i
REFINES entry_point

IMPORTS
    ctx, array

OPERATIONS
    op = set_array_value
END
```


This document is the property of CLEARSY.

The structure of the project should look like this using the *Top-Bottom Graphical view*:

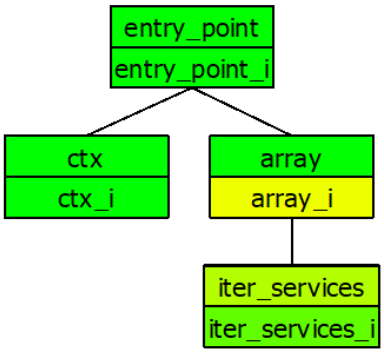


Figure 25: Expected output of the Top-Bottom Graphical view

6.7 Proving the implementation

As with previous exercises involving loops and arrays, not all proof obligations (POs) may be discharged automatically by the prover. This is especially true for invariants involving quantified expressions or properties about the maximum value. After running  , you may see some POs remain unproved:

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
array	OK	OK	2	1	1	OK
array_i	OK	OK	11	6	5	OK
ctx	OK	OK	0	0	0	OK
ctx_i	OK	OK	1	1	0	OK
entry_point	OK	OK	0	0	0	OK
entry_point_i	OK	OK	0	0	0	OK
iter_services	OK	OK	8	6	2	OK
iter_services_i	OK	OK	10	9	1	OK

Figure 26: Expected Atelier B main screen after running the automatic prover

This does not necessarily mean your code is incorrect. In formal methods, especially with loops and quantified invariants, it is common for the automatic prover to require additional guidance. You should manually inspect the generated proof obligations to ensure they are logically correct and reflect the intended properties of your algorithm.

To fully discharge all remaining proof obligations, you can use the `.pmm` proof files provided in the GitHub repository. Compare your machines with the reference versions from GitHub to ensure

consistency.

Then, place all `.pmm` files into your project directory. Then:

- Select all machines in your project (using `Ctrl+A` or Shift-click).
- Click the  button to run the automatic prover on all components.
- After the prover completes, select all machines that still have unproved proof obligations and click the  button to apply the user proofs from the `.pmm` files. This will discharge any remaining unproved proof obligations.
- Once all POs are proved, select each implementation model and use **Component** > **B0 Check** (or press `Ctrl+B`) to ensure they are ready for code generation.

Here is how this process appears in Atelier B:

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
ctx	OK	OK	0	0	0	-
ctx_i	OK	OK	1	1	0	-
entry_point	OK	OK	0	0	0	-
entry_point_i	OK	OK	3	3	0	-
iter_services	OK	OK	8	8	0	-
iter_services_i	OK	OK	10	10	0	-
max_array	OK	OK	2	2	0	-
max_array_i	OK	OK	36	36	0	-
max_array_r	OK	OK	2	2	0	-

Figure 27: Applying the User Pass and B0 check in Atelier B

6.8 Generating and testing the C code

Once all proof obligations are discharged and the implementation models are B0 checked, you can generate the corresponding C code using Atelier B's built-in translator.

The following files are generated by Atelier B: `entry_point.h`, `entry_point_i.c`, `array.h`, `array_i.c`, `iter_services.h`, `iter_services_i.c`, `ctx.h`, and `ctx_i.c`. In addition, you need to create `main.c` to compile and run the project.

```c
/* File: main.c */
#include "entry_point.h"
#include <stdio.h>
#include <stdint.h>
```

```
int main() {

    entry_point__INITIALISATION();

    printf("The initial content of the array is: ");
    for (int i = 0; i <= ctx__NN; i++) {
        printf("%d ", array__Array[i]);
    }
    printf("\n");

    entry_point__op();
    printf("The content of the array after calling 'set_array_value' operation is: ");
    for (int i = 0; i <= ctx__NN; i++) {
        printf("%d ", array__Array[i]);
    }
    printf("\n");
}
```

It initializes `entry_point`, prints the initial values of the array, calls the op operation, and then prints the updated values of the array.

Create a new Makefile, similar to the one provided in the previous exercises, to compile and run the code. Once the code compiled and successfully executed, an output similar to the following should be displayed:

```
The initial content of the array is: 0 0 0 0 0 0 0 0 0 0 0
The content of the array after calling 'set_array_value' operation is: 0 1 2 3 4 5 6 7
↪ 8 9 10
```

6.9 To go further

- **Filling a subarray with a given value**: In the `set_array_value` operation, add a new parameter `start_index` and `end_index`, which will define the range of indices to be filled with the given value. The loop should iterate only over the indices from `start_index` to `end_index`, inclusive. You shall verify that no boundary violations occur, such as `start_index < 0` or `end_index > NN`. The invariant must be updated accordingly to reflect these changes. `iter_services` may also need updates to support this new functionality.

This document is the property of CLEARSY.

7 Find the maximum of an array

7.1 Introduction

The following exercise focuses on another essential use of loops in B: **finding the maximum value in an array**. This is a classic problem in algorithm design that reinforces iteration, conditional logic, and the concept of maintaining state (in this case, the current maximum).

At the end of this exercise, you should:

- **Deepen your understanding of loop invariants**: How can we express that our current maximum is always greater than or equal to the elements we have already checked?
- **Apply comparison logic**: Learn how to compare and update values within a loop structure.
- **Practice iteration over arrays**: Reinforce your ability to traverse arrays using abstract iterators, a key tool in B for managing collections.

This exercise helps solidify core algorithmic thinking and introduces a common pattern used in many real-world systems: scanning through data to extract specific information using a verified loop.

7.2 Natural language description

Find the maximum value of a `NAT` **array** of size `11`, from `0` to `10`. Write an operation that finds and returns the maximum value of an array passed as parameter. You should iterate through each index using an abstract iterator, compare each value with the current known maximum, and update it when a higher value is found. The loop must include suitable invariant and variant expressions to ensure it is both correct and terminating.

7.3 Formal specification

Since **abstract iterators**, **arrays** and **modular architecture** have already been covered, proceed to the next step: **finding the maximum value of an array**.

In order to properly use the abstract iterator, you should create its specification and implementation models, as shown in the previous exercise.

You also need to create a context machine, alongside its implementation, which will contain the `NN` constant, representing the size of the array. The size of the array is `NN + 1`, so since there are 11 elements, `NN := 10`.

In the main machine, named `max_array`, define the abstract specification of the maximum value operation. It should contain the following clauses respecting their order:

This document is the property of CLEARSY.

1. SEES: to properly access the NN constant.
2. OPERATIONS: to define the operation getMaximum, which will compute the maximum value of the array. An array of type (0..NN) --> NAT must be passed as parameter. It may use the abstract operations max and ran to find the maximum value within the image of the array.

```
OPERATIONS
    maxi <-- getMaximum (array) =
    PRE
        array : (0..10) --> NAT
    THEN
        maxi := max(ran(array))
    END
END
```

Note that neither max or ran can be directly proved nor converted to C code, because they are considered abstract operations.

There are no VARIABLES or INVARIANT clauses in the abstract specification, since the operation does not need any concrete variables or invariants to be defined.

7.4 Refinements

Going straightforwardly to the implementation can be painful, and you may struggle with proofs. This method consists of breaking down the difficulty by introducing an intermediate refinement between the loop and its specification. The starting assumption is that the abstract machine models a high-level loop algorithm. If the substitution described in the specification is too abstract or synthetic, a refinement must be introduced to more explicitly model the behavior of the loop across all iterations.

During the refinement phase, we should replace some abstractions with a quantified expression. A quantified expression is a logical statement that uses quantifiers (such as “for all” or “there exists”) to express properties about a range of elements in a set or domain.

This way, a well-suited clause for the final invariant can be found, which will be used in the final implementation.

The refinement step must contain both SEES and OPERATIONS clauses, as well as the specification.

Here you have the refined OPERATIONS clause:

This document is the property of CLEARSY.

```
OPERATIONS
    maxi <-- getMaximum (array) =
    BEGIN
       maxi : (maxi : ran(array) & !(index).(index : dom(array) => array(index) <=
↪   maxi))
    END
END
```

The refinement proposes an algorithm that can be implemented in the final loop in a way that can be both **proved** and **translated to C code**: We loop through the array and check if the current value is greater than the maximum value found so far. If it is, we update the maximum value. The expression can be divided in two parts:

- `maxi : ran(array)`, which assures the maximum value is in the range of the array.
- `!(index).(index : 0..NN => array(index) <= maxi))`, which assures the maximum value is greater than all other values in the array. It is our quantified expression.

We may translate the expression to the mathematical domain in order to be able to fully prove it. Note that `!(index).(index : 0..NN => array(index) <= maxi)` is an extension of the max notation:

$\exists i_{\text{max}} \in [0, NN]$ **such that** $\forall i \in [0, NN],\ a[i] \leq a[i_{\text{max}}]$

Quantified expressions usually facilitate the job of automatic provers.

To put it in a nutshell, during the refinement phase, three steps should be distinguished:

1. First, express the loop algorithm (introduce a refinement step).
2. Then, take this algorithm and replace certain constants with the loop index.
3. Finally, the proof may allow for adjustments to the loop invariant if necessary.

7.5 Implementation

The implementation's structure of `max_array` still only needs a `SEES` and an `OPERATIONS` clause.

The implementation of `getMaximum` is a bit more delicate. In B, just like in most programming languages, when we want to go through all elements of an array to find its maximum, we use a loop. So how do we build this loop? Starting from the variant:

1. We plan to iterate from the beginning to the end of the array. Since our array goes from `0` to `NN`, we will make exactly `NN + 1` iterations.

This document is the property of CLEARSY.

2. To control the loop, we introduce a `flag` variable, which keeps true while there are elements to be treated.
3. Using the abstract iterator, each iteration will remove an element from the `Todo` set and add it to the `Done` set. The loop will continue until there are no more elements in the `Todo` set.
4. A natural choice for the variant may be `NN + 1 - card (Done)`, where `card (Done)` is the number of elements present on `Done`, used as our control loop index.
5. The newfound variant is a positive expression that decreases with each iteration, ensuring the loop will eventually terminate.

```
VARIANT
    NN + 1 - card (Done)
```

In order to make the final refinement, we may also find a right invariant. It can be done by replacing the `NN` constant on the quantified expression with a loop index variable. In each iteration, `maxi` is the local maximum value of the already processed part of the array.

For the purpose of giving a hand to the automatic prover, we may define a `local_maxi` variable. It will be used to store the local maximum of the array at each iteration, therefore `local_maxi` can be updated multiple times inside the loop. This variable is initialized to the first element of the array, which is `array(0)`. At the very end, `local_maxi` will contain the maximum value of the array, and we can simply copy its value to `maxi` via `maxi := local_maxi`.

Consequently, the invariant behaves as follows:

- Its step must be included in `0..NN`.
- `!(index).(index : 0..step => array(index) <= local_maxi))`.
- From the `xx := yy + 3` example, we learned that the invariant may also contain some types of loop variables.

Knowing from the previous exercise, very often, even with the best suited invariant, it is impossible to fully prove everything without interactive proof. Here you have a well suited invariant, taking into account our newest quantified expression:

```
INVARIANT
    // Variables' types
    step : 0..NN &
    local_maxi : ran(array) &
    array : (0..NN) --> NAT &
    Done <: 0..NN &
```


This document is the property of CLEARSY.

```
    Todo <: 0..NN &
    Done /\ Todo = {} &
    Done \/ Todo = 0..NN &
    continue = bool(not(Todo={})) &
    step : Done &

    // Local maximum properties
    !index.(index : Done => array(index) <= local_maxi)
```

Since all the pieces are on the table, both the variant and the invariant combined with our refinement machine, write the final implementation of the loop:

```
maxi <-- getMaximum (array) =
VAR local_maxi, step, flag IN
    // step: loop's index
    // local_maxi: local maximum value
    // flag: array bounds control

    // step := 0;
    flag <-- init_iter ;

    // We need to set local_maxi := array(0);
    flag, step <-- next_iter ;
    local_maxi := array(step);

    WHILE (flag = TRUE) DO
        VAR var_loc IN
            flag, step <-- next_iter ;
            var_loc := array(step);

            IF (local_maxi < var_loc)THEN
                local_maxi := var_loc
            END
        END
    INVARIANT
        // Variables' types
        step : 0..NN &
        local_maxi : ran(array) &
        array : (0..NN) --> NAT &
        Done <: 0..NN &
        Todo <: 0..NN &
```


This document is the property of CLEARSY.

```
        Done /\ Todo = {} &
        Done \/ Todo = 0..NN &
        flag = bool(not(Todo={})) &
        step : Done &

        // Local maximum properties
        !index.(index : Done => array(index) <= local_maxi)
    VARIANT
        NN + 1 - card(Done)
    END;
    maxi := local_maxi
END
```

`step` is the loop's index, `local_var` is the array's value at `step`, `local_max` is the maximum value in `0..step`, `continue` is our loop condition. `local_var` is voluntarily declared inside the loop to avoid characterizing it on the invariant.

The final structure of the project should look like this using the Top-Bottom Graphical view:

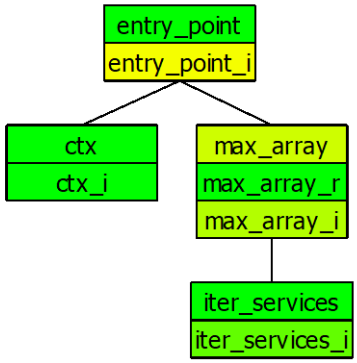


Figure 28: Expected output of the Top-Bottom Graphical view

As you have done so far, create an entry-point machine that imports both `max_array` and `ctx` to be the entry-point of the system. It must redefine `getMaximum`, specifying the type of the `maxi` return value. The entry-point machine specification should look like this:

```
MACHINE
    entry_point

OPERATIONS
    maxi <-- get_max (array) =
```

```
    PRE
        array : (0..10) --> NAT
    THEN
        maxi :: NAT
    END
END
```

In the implementation: `get_max` must call `getMaximum` to retrieve the maximum value from the array. The implementation of the entry-point machine should look like this:

```
IMPLEMENTATION entry_point_i
REFINES entry_point

IMPORTS max_array, ctx

OPERATIONS
    maxi <-- get_max (array) =
    BEGIN
        maxi <-- getMaximum(array)
    END
END
```

7.6 Proving the implementation

As you are used to, you can now run  . As you already know, some POs may be not proved automatically.

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
ctx	OK	OK	0	0	0	OK
ctx_i	OK	OK	1	1	0	OK
entry_point	OK	OK	0	0	0	OK
entry_point_i	OK	OK	3	1	2	OK
iter_services	OK	OK	8	6	2	OK
iter_services_i	OK	OK	10	9	1	OK
max_array	OK	OK	2	1	1	OK
max_array_i	OK	OK	36	27	9	OK
max_array_r	OK	OK	2	1	1	OK

Figure 29: Expected Atelier B main screen after running the automatic prover

Let's take a look at one of the unproved proof obligations (PO) generated by Atelier B:

This document is the property of CLEARSY.

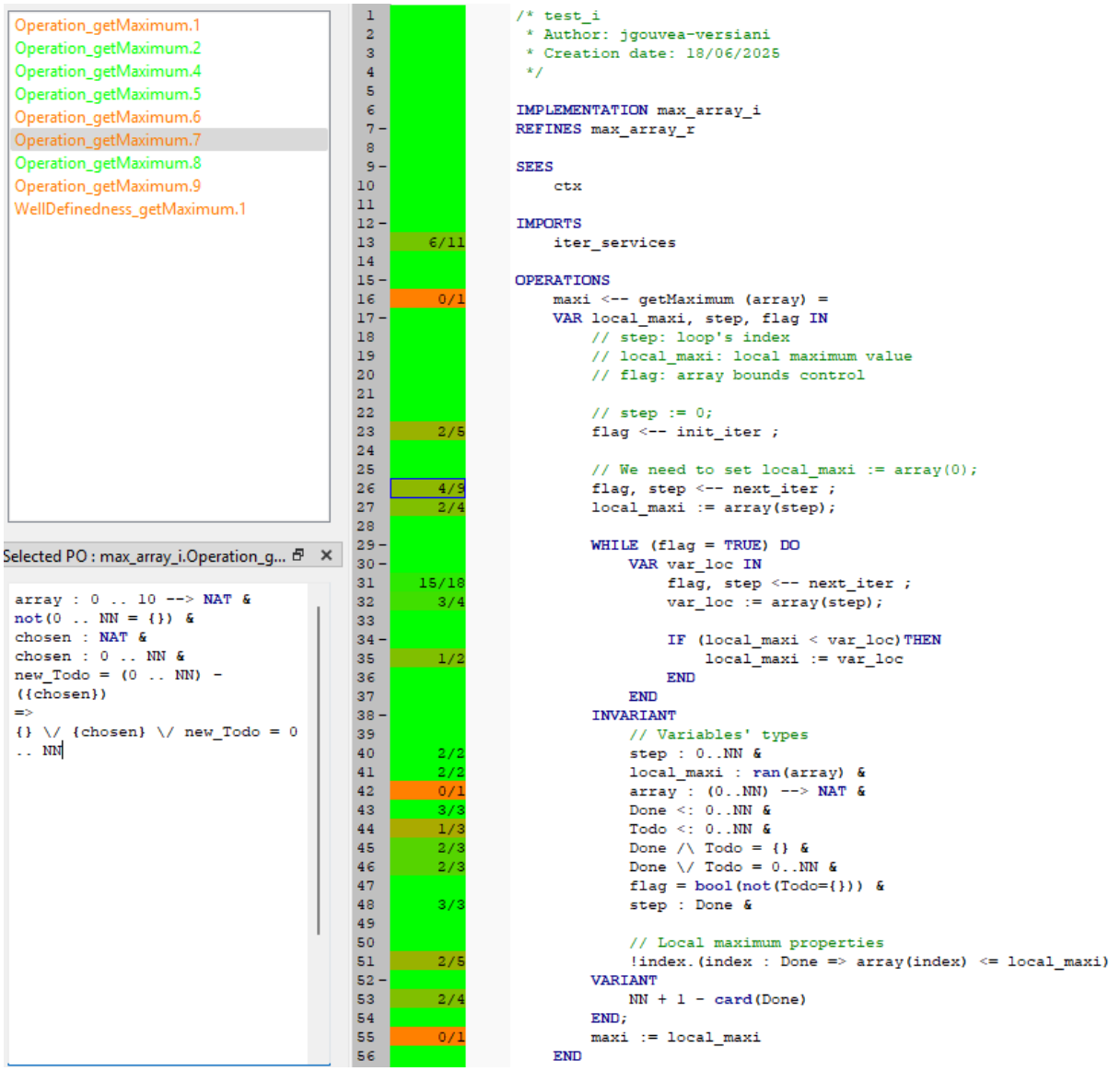


Figure 30: Unproved PO for the *Find the maximum of an array* exercise

For the sake of the example, we can manually inspect the generated proof obligations, and check if they seem correct. Let's take a look at the shown one:

```
array : 0 .. 10 --> NAT & not(0 .. NN = {}) & chosen : NAT & chosen : 0 .. NN &
 ↪  new_Todo = (0 .. NN) - ({chosen})
=>
{} \/ {chosen} \/ new_Todo = 0 .. NN
```

Note that `0..NN /= {}` is always true in our scenario since our array is not empty. Also, `array :`

`0 .. 10 --> NAT` is always true since we defined the array as a total function mapping indices from `0` to `NN` to natural numbers (NAT).

This proof obligation states that if `chosen : 0 .. NN` and `new_Todo = (0 .. NN) - ({chosen})` is true, then `{} \/ {chosen} \/ new_Todo = 0 .. NN` must also be true. If `new_Todo = (0 .. NN) - ({chosen})`, knowing that `chosen` is an element of `0 .. NN`, it means that `new_Todo` contains all elements of `0 .. NN` except for `chosen`. Therefore, `{} \/ {chosen} \/ new_Todo = 0..NN`. The automatically generated PO seams correct, although it may not be proved automatically.

To fully discharge all remaining proof obligations, you can use the `.pmm` proof files provided in the GitHub repository. Compare your machines with the reference versions from GitHub to ensure consistency.

Then, place all `.pmm` files into your project directory. Then:

- Select all machines in your project (using `Ctrl+A` or Shift-click).
- Click the  button to run the automatic prover on all components.
- After the prover completes, select all machines again and click the  button to apply the user proofs from the `.pmm` files. This will discharge any remaining unproved proof obligations.
- Once all POs are proved, select each implementation model and use **Component** > **B0 Check** (or press `Ctrl+B`) to ensure they are ready for code generation.

Here is how this process appears in Atelier B:

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
ctx	OK	OK	0	0	0	OK
ctx_i	OK	OK	1	1	0	OK
entry_point	OK	OK	0	0	0	OK
entry_point_i	OK	OK	3	3	0	OK
iter_services	OK	OK	8	8	0	OK
iter_services_i	OK	OK	10	10	0	OK
max_array	OK	OK	2	2	0	OK
max_array_i	OK	OK	36	36	0	OK
max_array_r	OK	OK	2	2	0	OK

Figure 31: Applying the User Pass and B0 check in Atelier B

This document is the property of CLEARSY.

7.7 Generating and testing the C code

To build and test the generated C code, ensure your project directory contains all the necessary files. This includes the generated files (`entry_point.h`, `entry_point_i.c`, `max_array.h`, `max_array_i.c`, `iter_services.h`, `iter_services_i.c`, `ctx.h`, `ctx_i.c`).

Additionally, you need to create a `main.c` file in your project directory and copy the following code into it. Also, add a Makefile similar to the one provided in the previous exercises to compile and run the code.

```c
/* File: main.c */
#include "entry_point.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
    srand(42);

    entry_point__INITIALISATION();

    int32_t max_array__array[ctx__NN + 1] = {0};
    for (int i = 0; i <= ctx__NN; i++) {
        max_array__array[i] = rand() % ctx__NN;
    }

    printf("Finding the maximum value in an array of integers: ");
    for (int i = 0; i <= ctx__NN; i++) {
        printf("%d ", max_array__array[i]);
    }
    printf("\n");

    int32_t maxi;
    entry_point__get_max(max_array__array, &maxi);
    printf("The maximum is: %d\n", maxi);
}
```

It initializes the `entry_point`, defines an array of size 11 full of random numbers then print its values, and then calls the `get_max` operation to retrieve the maximum value from the array.

Create a new Makefile, similar to the one provided in the previous exercises, to compile and run the code. Once the code compiled and successfully executed, an output similar to the following should be displayed:

This document is the property of CLEARSY.

```
Finding the maximum value in an array of integers: 6 0 1 1 2 8 1 0 5 3 4
The maximum is: 8
```

7.8 To go further

- **Finding the minimum value in an array**: Create a new operation `getMinimum` operation to find the minimum value instead of the maximum. This can be done by changing the comparison logic in the loop.
- **Finding the local maximum in the array**: Add two parameters to the `getMaximum` operation, `start` and `end`, to find the local maximum value within `[start, end]` indexes of the array. You must change the loop invariant and variant accordingly to ensure the loop iterates only over the specified range. The invariant should ensure that the local maximum is greater than or equal to all values in the specified range.

This document is the property of CLEARSY.

Appendix

Installing ProB2-UI

ProB2-UI installer is available for Windows, macOS, and Linux. You can download it from the ProB2-UI website, on the `ProB2-UI (based on JavaFX)` section.

Installing gcc and make on Linux, macOS, and Windows

This guide explains how to install gcc (GNU Compiler Collection) and Make, on various operating systems.

Linux (Ubuntu/Debian)

1. Update package list.

```
sudo apt update
```

2. Install gcc and make.

```
sudo apt install build-essential
```

- `build-essential` includes gcc, make, and other necessary tools.

3. Verify installation.

```
gcc --version
make --version
```

Linux (Fedora)

1. Update package list.

```
sudo dnf update
```

2. Install development tools.

```
sudo dnf groupinstall "Development Tools"
```

3. Verify.

This document is the property of CLEARSY.

```
gcc --version
make --version
```

macOS

1. Install Xcode Command Line Tools.

```
xcode-select --install
```

2. Verify installation.

```
gcc --version
make --version
```

- Xcode Command Line Tools include both gcc (an alias to Clang) and make.

Windows (Install via WSL)

1. Install WSL. In PowerShell (Admin mode):

```
wsl --install
```

2. Install the latest version of Ubuntu in the Microsoft Store.
3. Open Ubuntu and follow the instructions for Ubuntu above.

This document is the property of CLEARSY.

Project configuration

We need to configure Atelier B to display the proof obligation (PO) generated and locate the one that are not proved.

1. Click on **Atelier** > **Preferences**.
2. Go to the Internal Editor tab and tick all Proof information options.

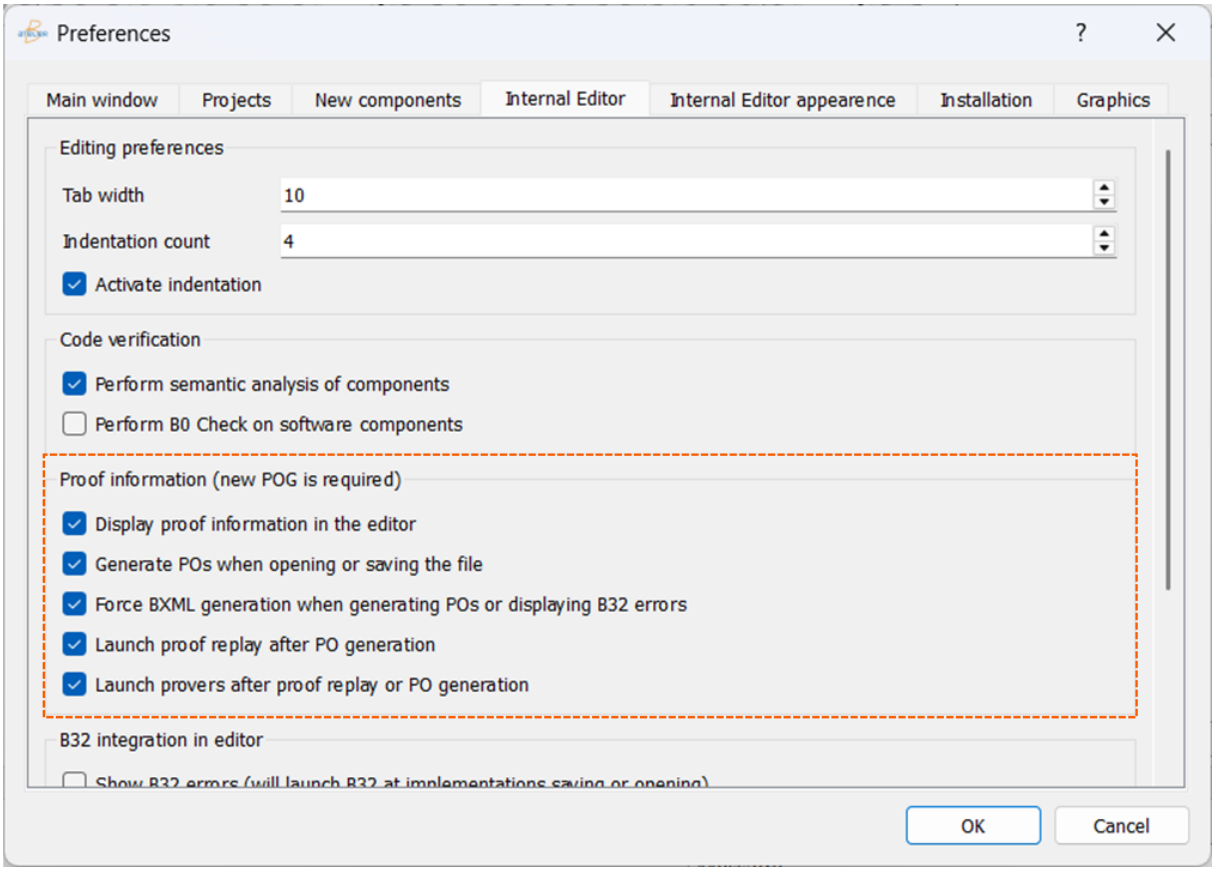


Figure 32: Preferences

In order to prove your code, Atelier B must generate proof obligations. We need to ensure the project use the latest version of the proof obligation generator :

- 1. Right-click on your project in the left panel, then click on Properties.

This document is the property of CLEARSY.

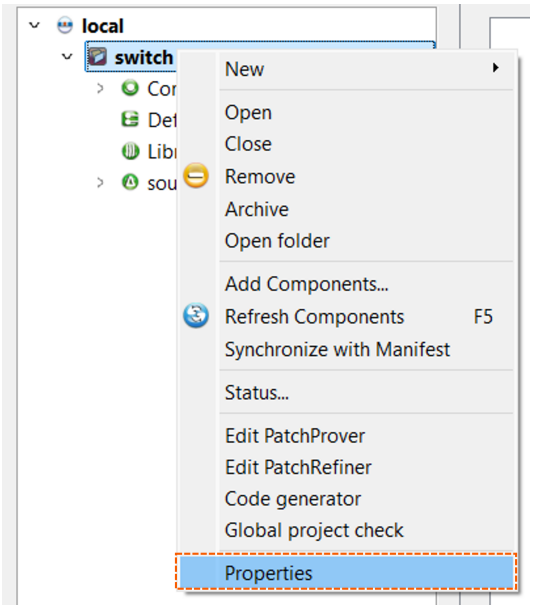


Figure 33: Properties

- 2. Go to the system modeling tab.

This document is the property of CLEARSY.

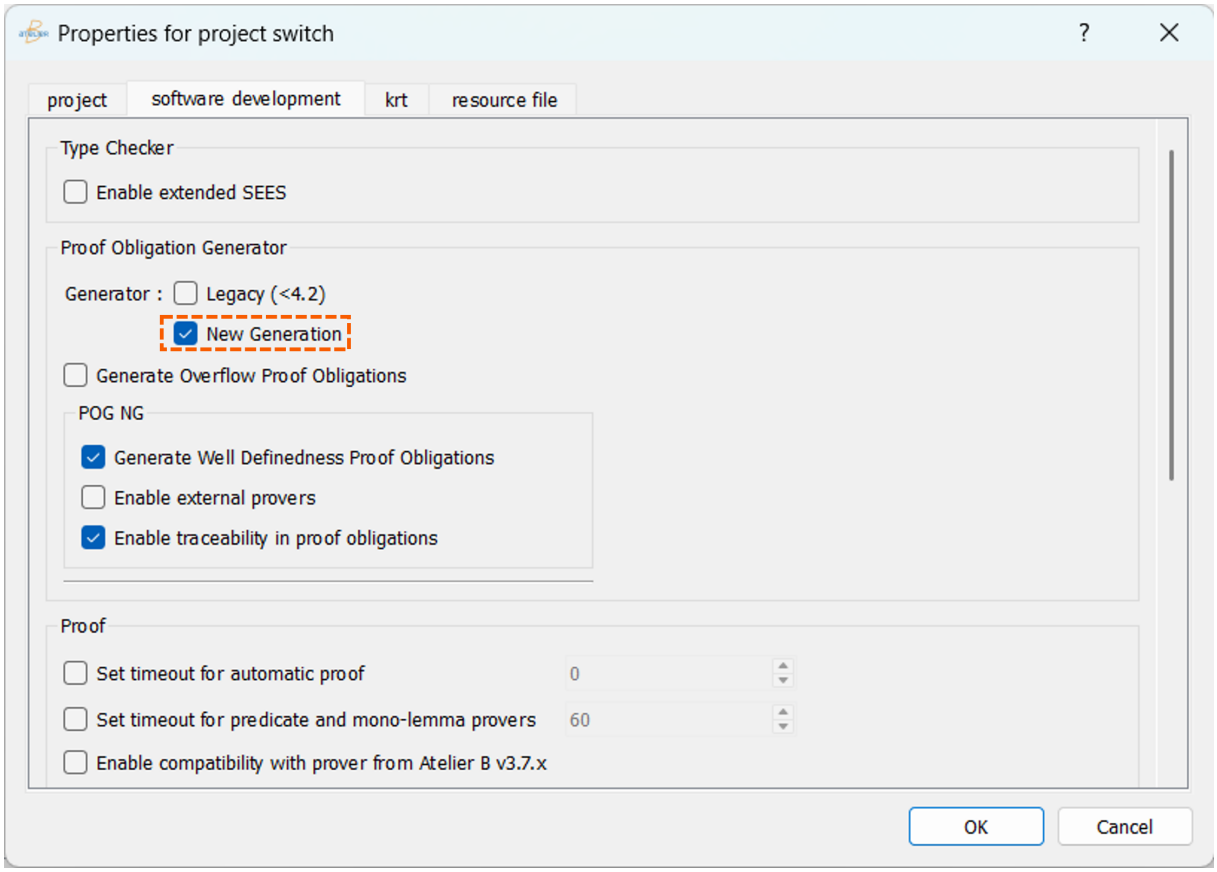


Figure 34: Software development

- 3. Ensure that "New Generation" is ticked in proof obligation generator.

Once you finally reach this step, you are all set to create, prove and generate your first software with atelier b.

This document is the property of CLEARSY.

Substitutions cheat sheet

The following table serves as a quick-reference guide to support your work throughout this workbook. It summarizes the **substitutions** in Atelier B, including their **syntax** and **applicability in B and B0 machines**.

For detailed explanations and additional information about each substitution, please refer to the **B Language Reference Manual**[7] available in Atelier B. You can access the manual by opening **Help** > **References** > **B Language Reference** in Atelier B. The manual is organized into chapters, and each item on this cheat sheet includes a reference to the relevant chapter in the last column for your convenience.

Substitution	Description	B	B0	Ref
$x := e$	Assign a value	✓	✓	6.3
$x :: E$	Assign from a set	✓		6.12
$x : (P)$	Assign satisfying predicate (P)	✓		6.13
S_1 `||` S_2	Simultaneous substitution	✓		6.18
`skip`	Do nothing	✓	✓	6.2
`BEGIN` S `END`	Group substitutions	✓	✓	6.1
`PRE` P `THEN` S `END`	Precondition	✓		6.4
`IF` P `THEN` S_1 `ELSE` S_2 `END`	Branch on condition	✓	✓	6.7
`CASE` x `OF ... END`	Multi branch on condition	✓	✓	6.9
`CHOICE` S_1 `OR` S_2 `END`	Nondeterministic choice	✓		6.6
`SELECT` P_1 `THEN` S_1 `... END`	Guarded choice	✓		6.8
`ANY` x `WHERE` P `THEN` S `END`	Declare local variable	✓		6.10
`ASSERT` P `THEN` S `END`	Assert property	✓	✓	6.5
S_1 `;` S_2	Sequence statements		✓	6.15
`VAR` x `IN` S `END`	Declare local variable		✓	6.14
`WHILE` P `DO` S `... END`	Loop while P is true		✓	6.17
`x ← op(...)`	Call operation		✓	6.16

[7] CLEARSY, *B Language Reference Manual*, https://www.atelierb.eu/wp-content/uploads/2023/10/b-language-reference-manual.pdf.

This document is the property of CLEARSY.

Bibliography

Abrial, Jean-Raymond. *The b-Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press, 1996. https://doi.org/10.1017/CBO9780511624162.

CLEARSY. *B Language Reference Manual*. Version 1.8.10. Aix-en-Provence, France, n.d. https://www.atelierb.eu/wp-content/uploads/2023/10/b-language-reference-manual.pdf.

———. *Installation Guide Community Edition*. CLEARSY, 2024. https://www.atelierb.eu/wp-content/uploads/2024/11/installation-guide-community.pdf.

Enderton, H. B. *A Mathematical Introduction to Logic*. 2nd ed. San Diego: Academic Press, 2001.

Lecomte, Thierry. "Event Driven b: Methodology, Language, Tool Support, and Experiments." *ResearchGate*, 2002.

Marcel, Oliveira, and Thierry Lecomte. "The b-Method [MOOC]," n.d. https://mooc.imd.ufrn.br/course/the-b-method.

Schneider, Steve. *The b-Method: An Introduction*. London, UK: Red Globe Press, Palgrave Macmillan UK, 2001. https://www.palgrave.com/gp/book/9780333792841.

———. *The b-Method: An Introduction*. Cornerstones of Computing. Palgrave Macmillan, 2001.

This document is the property of CLEARSY.