Block

Building a Complete Software B Project

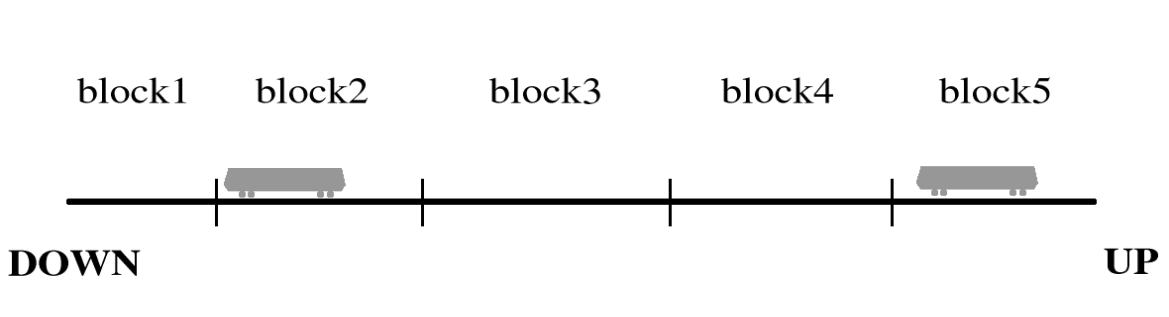
Presentation

This exercise is an example of how B can be used to develop a piece of software. It is based on a real example, although it has been simplified to fit the size of an exercise. This software controls a railroad line, divided into fixed blocks. The purpose of the functionality developed here is to manage safely block occupancy by trains.

- This guide first briefly presents the system.
- Then it presents the software general architecture and gives the detailed requirements.
- In the practical part, we focus on how B is used to formalize those requirements into abstract software specification and especially how set notations turn B into a high level programming language. We also focus on how abstract properties may strenghen the formal model.
- Eventually, starting from the abstract formal level, we shall present the principle of producing the detailed design level to build step by step implementable code. We shall also explain how the whole B software part is integrated into the whole software.

System Description

A railroad line is supposed to be divided into fixed blocks. The line has two directions 'up' and 'down'. Each block may only be connected to an upward block and to a downward block. So the line is quite simple, since it has no switch. Figure 1 gives an example of such a line with 5



blocks. Actually, this system is the simplification of a more realistic one handling switches.

Figure 1: A Railroad Line of 5 Blocks

Trains may drive upbound or downbound on the railroad line and they may change direction at any time.

The purpose of the functionality developed in this exercise is to establish safely, from the software point of view, which blocks are occupied by a train, and which are free. Here is the basic principle given by the system analyses. For each block a detector located along the track called "Trackside Detector" is used to detect trains. A train is equipped with an antenna

located below the car. When the antenna is above a trackside detector, a signal inside the detector is produced, so the train presence may be detected. Now building the software appears to be easy, since we just need to read for each block the state of its trackside detector. However, this solution raises two issues, both related with safety:

1. The information given by a trackside detector is not accurate enough on the border of detectors.
2. Trackside detectors (or antennas) may be faulty.

To overcome those issues, the following elements are added to the system specification.

- A border detector is used at each borderline between two blocks to achieve accurate block occupancy detection. When a block trackside detector or border detector is occupied then the block is considered occupied.
- Exit detectors located after a block border (in the upward block or in the downward block) are used to detect trains leaving the block. A block is considered to be released on the falling edge of one of its exit detectors.
- The Trackside Detector Loss (TDL) alarm is set on a block when a trackside detector inconsistency happens. When a block trackside detector read value is "free" although it should be "occupied". When a TDL alarm is set, the procedure to release it requires that an operator at the command center should send back an alarm acknowledgement.
- To avoid unjustified TDL alarm due to the lack of accuracy of trackside detectors, blocks may be masked for TDL alarm when trains are located at the block border.

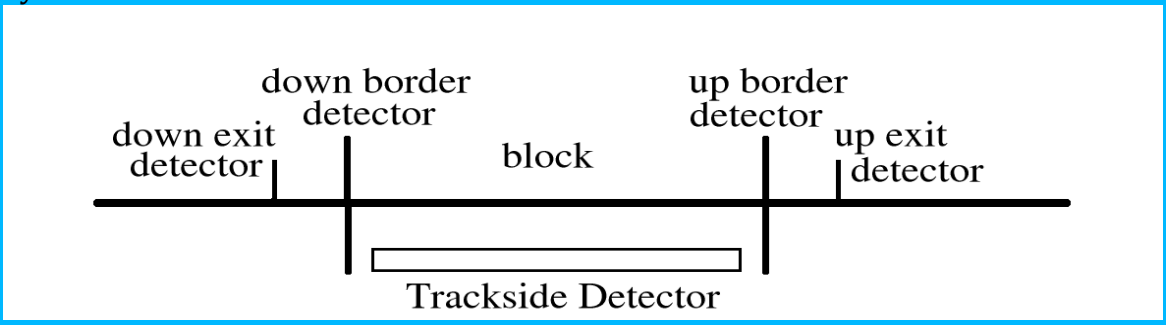


Figure 2: Different Types of Detectors Related to a Block

This system description is not complete, but it gives the basic principles of the system. The main input of the exercise is the software requirements.

Software Requirements

Architecture

The software developed in B is a subpart of the whole software, containing the safe high-level part. Its entry point should be an operation called execute_cycle, which is launched at a regular pace. The B software can be described by a top-down functional decomposition. In this example, the entry module, called main, calls in order from the left to the right its submodules: inputs, blocks_occupancy, other_module and outputs (for simplification purpose, the last modules are not described here).

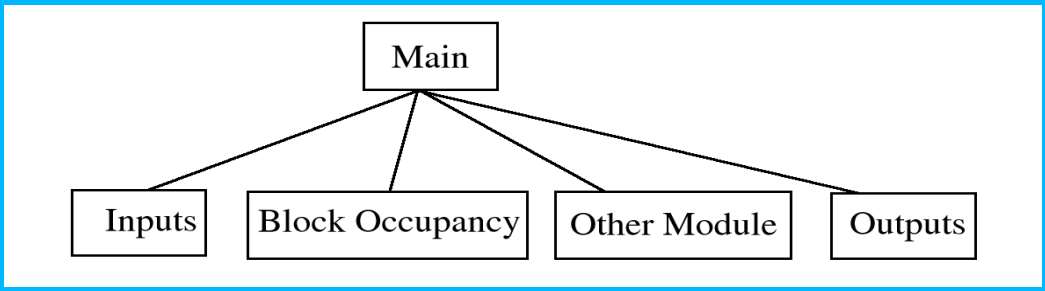


Figure 3: Functional Architecture

Types and configuration data
(see appendix)

Inputs
(see appendix)

Blocks Occupancy Sequence
(see appendix)

Blocks Occupancy
(see appendix)

Completing the Abstract Formal Level (Practical Part)

The abstract formal level of a B software project is the part where the requirements are formalized in B. In this example the abstract formal level is in part already written. Actually, as this B architecture follows the functional decomposition described in the software requirements and as the static data are mostly already defined by the software requirements, this part is quite straightforward. To complete the abstract formal model, you have to formalize operation bodies.

Launch Atelier B and open the "block" project (the skeleton is available at https://github.com/CLEARSY/hackathon-2024).

The following components are part of the abstract formal level. Let's have a look at each of them and especially let's read their description in natural language given inside comments.

- **configuration**:
  This abstract machine specifies the types and the line configuration data used by the rest of the project. See clauses 'SETS', 'CONCRETE_CONSTANTS' and 'ABSTRACT_CONSTANTS'.
- **inputs**:
  This abstract machine specifies data extracted from messages coming from other equipments. See clause 'ABSTRACT_VARIABLES'. The values of those variables are set at the beginning of each execute cycle, by calling operation 'read_inputs'.
- **main**:
  Entry point of project 'block'. It contains one operation, execute_cycle. Complete the specification by adding the property indicated in natural language.
- **main_i**:
  Implementation of main.
- **block_occupancy**:
  This abstract machine specifies block occupancy with 3 variables ('occupied_blocks', 'tdla', 'masked_blocks') and 6 operations.

Complete the operation bodies, according to their specification in natural language. Start by specifying the bodies of operations: 'release_tdl_alarm', 'set_tdl_alarm' and 'occupy_blocks'.
Some definitions are given in the 'DEFINITIONS' clause that you may find helpful. After writing an operation type check it.

- **block_occupancy_seq**:
  This abstract machine is the top level specification of the block occupancy functionality. The two variables 'occupied_blocks' and 'tdl_alarm' are repeated and one operation 'set_block_occupancy' is defined to sequence all the treatments specified in block_occupancy.
  Complete the body of operation 'set_block_occupancy' by specifying that the two variables are transformed so that their invariant is true and safety property 1 is true.
- **block_occupancy_seq_i**:
  This is the implementation of the abstract machine 'block_occupancy_seq'. That implementation imports abstract machine 'block_occupancy'. Complete the operation body of operation 'set_block_occupancy' according to its specification in natural language. This operation is part of the abstract formal level, although it is an implementation, since the software specification of the block occupancy functionality specifies the 6 operations and their sequencing in the top-level operation.
  Properties formalized in sequencing abstract machines are of high importance, since they strengthen the whole model. Usually, those properties come from system analyses. When a property is formalized at the model highest level, we are sure that the whole software complies with this property. In our example, operation 'occupy_blocks' is built to comply with property 1, but it is not sure that operation 'set_block_occupancy', that calls 'occupy_blocks', also comply with property 1, since it also calls other operations that modify occupied blocks. We can be sure of this by specifying operation 'set_block_occupancy', so that it respects property 1. Then if all the proof obligation of its implementation are demonstrated, it means that property 1 is indeed respected.

When the abstract formal level is complete and is successfully type checked, launch automatic proof in Force 0 and in Force 1, then use User Pass (they are several pmm files on the github directory). If some proof obligation still remain not proved, try automatic User Pass, which is configured on this project to call the predicate prover.

Detailed Design Level

Once the Abstract Formal Level is complete and proved the next step is to develop the Detailed Design Level, so that every abstract machine should be implemented.
In this project abstract machines 'configuration' and 'inputs' are supposed to be basic machine, so their implementation is directly written in programming language. This is how to interface code translated from B with code not handled in B.
Abstract machine 'blocks_occupancy' is completely implemented in B, by implementation 'block_occupancy_i' and several submodules (including abstract iterator). To write those submodules, software requirements are no longer used. We start from abstract machine 'block_occupancy' and we refine it step by step using some refinement tactics.
For instance, abstract variable:

```
occupied_blocks <: t_block
```

is refined by concrete variable:

```
occupied_blocks_i : t_block --> BOOL
```

(this concrete variable is translated by an array), so that all the occupied blocks are associated to TRUE and non occupied blocks are associated to FALSE. Formally, the following linking invariant is used:

occupied_blocks = occupied_blocks_i~[{TRUE}]

Union and set difference of blocks are refined by loops on each element of t_block.

Appendix: Software Requirements

Types and configuration data

Constant Name:
t_block
Typing:
t_block <: t_block_i
Description:
Set of all blocks of the line

Constant Name:
t_border
Typing:
t_border <: t_border_i
Description:
Set of all borders of the line

Constant Name:
t_exit
Typing:
t_exit <: t_exit_i
Description:
Set of all exit detectors of the line

Constant Name:
cfg_block_to_block_up
Typing:
cfg_block_to_block_up : t_block +-> t_block
Description:
Function giving for a block the next upward block. An upward terminal block does not have any next upward block.

Constant Name:
cfg_block_to_block_down
Typing:
cfg_block_to_block_down : t_block +-> t_block
Description:
Function giving for a block the next downward block. A downward terminal block does not have any next downward block.

Constant Name:
cfg_block_to_border_detector_up
Typing:
cfg_block_to_border_detector_up : t_block +-> t_border
Description:
Function giving for a block its upward border. An upward terminal block does not have any upward border.

Constant Name:
cfg_block_to_border_detector_down
Typing:

cfg_block_to_border_detector_down : t_block +-> t_border

Description:

Function giving for a block its downward border. A downward terminal block does not have any downward border.

Constant Name:

cfg_block_to_exit_detector_up

Typing:

cfg_block_to_exit_detector_up : t_block +-> t_exit

Description:

Function giving for a block its upward exit detector. An upward terminal block does not have any upward exit detector.

Constant Name:

cfg_block_to_exit_detector_down

Typing:

cfg_block_to_exit_detector_down : t_block +-> t_exit

Description:

Function giving for a block its downward exit detector. A downward terminal block does not have any downward exit detector.

Inputs

Description:

Management of input messages

- 1 message from trackside equipements (trackside detectors, border detectors, exit detectors)
- 1 message from the Control Center (initialized blocks, TDC alarm acknowledgement)

Variable Name:

occupied_trackside_detectors

Typing:

occupied_trackside_detectors <: t_block

Description:

Set of blocks having an occupied trackside detector

Initialization:

occupied_trackside_detectors := t_block

Variable Name:

occupied_border_detectors

Typing:

occupied_border_detectors <: t_border

Description:

Set of occupied trackside detector

Initialization:

occupied_border_detectors := t_border

Variable Name:

occupied_exit_detectors

Typing:

occupied_exit_detectors <: t_exit

Description:

Set of occupied exit detector
Initialization:
occupied_exit_detectors := t_exit

Variable Name:
occupied_exit_detectors_prev
Typing:
occupied_exit_detectors_prev <: t_exit
Description:
Set of occupied exit detector during the previous execute cycle
Initialization:
occupied_exit_detectors_prev := t_exit

Variable Name:
cc_init
Typing:
cc_init <: t_block
Description:
Set of blocks beeing initialized by the Control Center
Initialization:
cc_init := {}

Variable Name:
cc_tdl_acknowledge
Typing:
cc_tdl_acknowledge : BOOL
Description:
Acknowledgement of TDL alarm for all blocks, send by the Control Center
Initialization:
cc_tdl_acknowledge := FALSE

Operation Name:
read_inputs
Input parameters:
none
Output parameters:
none
Functionality:
This function reads the inputs messages.

Operation Name:
save_inputs_prev
Input parameters:
none
Output parameters:
none
Functionality:
This function saves the actual value of occupied exit detectors for the next execute cycle.

Blocks Occupancy Sequence

Description:
Management of block occupancy at sequencing level

Variable Name:
occupied_blocks
See:
block_occupancy

Variable Name:
tdl_alarm
See:
block_occupancy

Operation Name :
set_block_occupancy
Input parameters:
none
Output parameters:
none
Functionality:
This function manages block occupancy, by calling operations in the following order:
- unmask_blocks
- release_tdl_alarm
- set_tdl_alarm
- mask_blocks
- release_blocks
- occupy_blocks
Properties:
- Property 1:
A block having one of its border detector occupied or having its trackside detector occupied has to be occupied.

Blocks Occupancy
Description:
Management of block occupancy and Trackside Detector Loss (TDL) alarm

Variable Name:
occupied_blocks
Typing:
occupied_blocks <: t_block
Description:
Set of blocks considered as occupied
Initialization:
occupied_blocks := t_block

Variable Name:
masked_blocks
Typing:
masked_blocks <: t_block

Description:
Set of masked blocks for TDL alarm
Initialization:
masked_blocks := {}

Variable Name:
tdl_alarm
Typing:
tdl_alarm <: t_block
Description:
Set of blocks in Trackside Detector Loss (TDL) alarm
Initialization:
tdl_alarm := t_block

Operation Name :
mask_blocks
Input parameters:
none
Output parameters:
none
Functionality:
This function unmasks some blocks (for TDL alarm). Blocks which do not become unmasked remain unchanged.
A block is unmasked when the block is free or when all of the following conditions are true:
1) The upward block has a free trackside detector or the upward block is free.
2) The downward block has a free trackside detector or the downward block is free.

Operation Name:
release_tdl_alarm
Input parameters:
none
Output parameters:
none
Functionality :
This function releases Trackside Detector Loss (TDL) alarm.
When a TDL alarm acknowledgment is received from the Control Center, then TDL alarm is released for all blocks.
If no TDL alarm acknowledgment is received then TDL alarm remains unchanged.

Operation Name:
set_tdl_alarm
Input parameters:
none
Output parameters:
none
Functionality :
This function sets Trackside Detector Loss (TDL) alarm. When a block does not become in TDL alarm, then the alarm remains unchanged.
A block becomes in TDL alarm, when the following conditions are true:
1) The block is occupied.

2) The block is not masked.
3) The block trackside detector is free.

Operation Name:
mask_blocks
Input parameters:
none
Output parameters:
none
Functionality:
This function mask some blocks (for TDL alarm).
A block is masked when the folowing conditions are true:
1) The block is not in TDL alarm.
2) One of the block borders is occupied.
Blocks which do not become unmasked remain unchanged.

Operation Name:
release_blocks
Input parameters:
none
Output parameters:
none
Functionality:
This function manages released blocks. Blocks which are not released remain unchanged.
A block is consider to be released when the following conditions are true:
1) The block is not in TDL alarm or the block is beeing initialized by the Control Center.
2) A block exit detector, which was occupied during the previous cycle, is now released.

Operation Name:
occupy_blocks
Input parameters:
none
Output parameters:
none
Functionality:
This function manages occupied blocks. Blocks which do not become occupied remain unchanged.
A block is consider to be occupied when one of its border detector is occupied or when its trackside detector is occupied.