Utilizing Rail Traffic Control Simulator in Verified Software Development Courses

Štefan Korečko(✉)

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Košice, Slovakia
stefan.korecko@tuke.sk

Abstract. With the increasing dependency of our society on automated systems, their correctness is of uttermost importance. Formal methods for software development, such as the B-Method, belong to rigorous approaches that may ensure the correctness. They offer mathematical apparatuses to prove that the software under development meets the corresponding requirements. But the need to comprehend such apparatus makes formal methods unpopular with students. They may not see the reasons why to use them. And many formal method courses do not include executable software development or the software developed is not used in an appropriate environment. Both problems are addressed by the TD/TS2JC toolset, described in this chapter. The toolset provides an appropriate virtual railway environment, where verified software controllers can run. The controllers can be developed with any formal method that offers translation to the Java programming language. The chapter also describes two of several control interfaces the toolset supports. It also introduces a compact, four to six hour long, course on verified software development with the B-Method, which utilizes the toolset.

Keywords: verified software · formal methods · B-Method · virtual environment · course · teaching · railway

1 Introduction

One of the well-recognized approaches to the development of correct software systems is the utilization of formal methods (FMs) for their specification and verification. FMs are rigorous mathematically based techniques for the specification, analysis, development and verification of software and hardware. Rigorous means that a formal method provides a formal language with unambiguously

This chapter is a result of the implementation of the Erasmus+ Key Action 2 project No. 2017-1-SK01-KA203-035402: “Focusing Education on Composability, Comprehensibility and Correctness of Working Software”.

defined syntax and semantics and mathematically based means that some mathematical apparatus (formal logic, set theory, etc.) is used to define the language.

The *B-Method* [1,2,11,15] is a state based, model-oriented formal method, intended for verified software development. It is one of the few software-related FMs that is used commonly in industrial practice, primarily in the railway sector. In this area, it is utilized for the safety-critical software behind automated urban metro subway systems [5]. The strength of the B-Method lies in a well-defined development process, which allows to specify a software system as a collection of components, called abstract machines, and to refine such an abstract specification to a concrete, implementable one. The concrete specification can be automatically translated to ADA, C, Java or another programming language. An internal consistency of the abstract specification and correctness of each refinement step are verified by proving a set of predicates, called proof obligations (PObs). The whole development process, including proving, is supported by Atelier B [18], an industrial-strength software tool.

A significant challenge in teaching formal methods for software development, including the B-Method, is to design a corresponding course in such a way that students will be able to develop a working piece of software using the method. The problem is rooted in the limitations of formal method languages. These languages usually cover basic constructs only, such as assignments, compositions, operations and operation calls, conditional statements and loops. The interaction with the user is no supported at all or limited to the console level. The situation gets even more complicated if one wishes to use appropriate examples, clearly showing the benefits of FMs, as advocated in [12–14].

To deal with this challenge, we developed the *TD/TS2JC* software toolset, which provides a virtual environment for programs, developed by students using formal methods. The toolset consists of a modified version of the *Train Director* [17] simulation game and an application, called *TS2JavaConn*, which allows using separately developed software controllers with the game. The controllers are Java programs that control switches and signals in railway scenarios, simulated by the game. The interface of the control programs can be configured, so the toolset is suitable for various formal methods. There is only one requirement the formal method has to fulfill: the existence of a compiler from its language to Java. And because the controllers are in Java, the toolset can be also used in situations that don't involve formal methods at all.

While the previous works [9,10] presented the toolset and its utilization in B-Method courses in general, here we discuss both of these topics in more depth. Section 2 describes the components of the TD/TS2JC toolset, their communication and usage. Section 3 presents and explains two different controller configurations and corresponding Java controllers. Section 4 presents a compact B-Method course, which utilizes the toolset. The total duration of the course is estimated to four to six hours, so it is ideal for special events, such as summer schools. The chapter concludes with an evaluation of a particular run of the course in Sect. 5.

2 TD/TS2JC Toolset

It is no surprise that rail traffic control systems, primarily those related to signaling [5], are one of the most successful domains of formal methods utilization. This can be attributed to two factors. First, the movement of trains is limited by tracks, which makes an automated control of their operation much easier than, for example, that of the road vehicles. Second, the railway is used to transport large number of passengers and goods at once, so individual accidents may have more severe consequences than those of other types of vehicles. Therefore, methods providing means to ensure the correctness of these control systems should be used. This special position of the rail traffic control systems was the primary reason why we decided to use virtual environments inspired by such systems in our FMs course. The decision was reinforced by the fact that the topic of our course is the B-Method and the B-Method played a key role in the verified development of the railway control software [5].

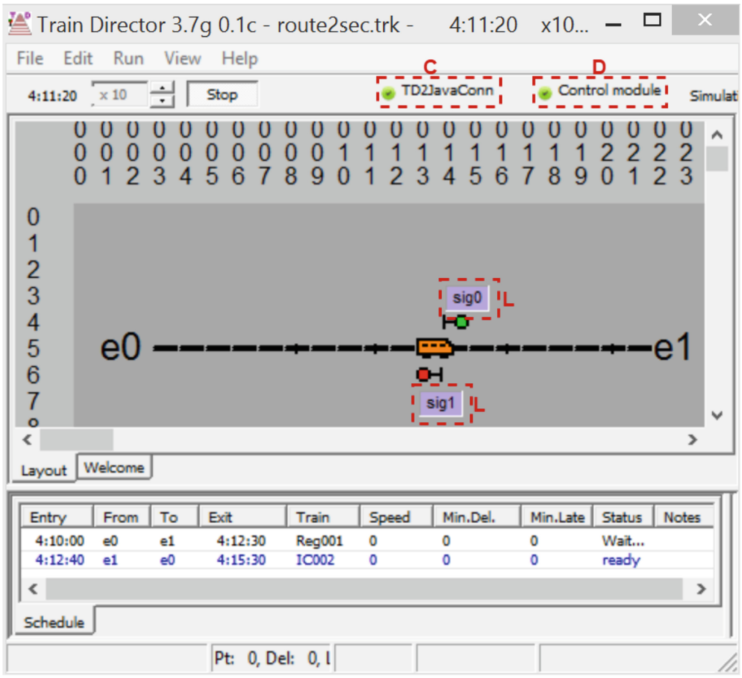
In order to provide virtual rail traffic control environments, we developed the *TD/TS2JC toolset*, consisting of two software applications. The first one is a modified version of an already existing simulation game, called *Train Director* (*TD*) [17]. The second one is *TS2JavaConn*, a newly developed Java application. The virtual environments are railway scenarios, simulated in TD. Signals and switches in these scenarios are managed by controllers (*control modules*) developed by students in the B-Method and translated to Java. TS2JavaConn serves as a proxy between TD and the control module. It listens to events occurring in the simulated scenario and executes methods of the control module accordingly. Subsequently, it informs TD about changes that should be applied to the scenario. Each control module is accompanied by a configuration file that defines how the events are translated to the method calls.

From the beginning of the toolset development, our goal was to provide a solution that is not limited to the B-Method. This is why we do not deal with formal methods at all in this section. The same is true for Sect. 3, where the control modules are presented on the Java language level only. The utilization in a formal methods course is shown in Sect. 4. The toolset, together with a set of examples, is available at [6].

2.1 Train Director

Train Director (*TD*) is a computer game, which simulates the work of the rail centralized traffic control (CTC). A railway scenario in TD consists of a track layout (track plan) and a train schedule. The player's task is to manipulate the signals and switches in the scenario in such a way that the trains arrive and depart according to the schedule. In TD/TS2JC, which uses the version 3.7 of TD, the task is carried out by the control module. This required several modifications of the simulator.

The first modification was a removal of those control mechanisms that should be implemented in control modules. The removed mechanisms, for example, prevented trains from colliding or entering the same section at once. As the removal enabled train crashes, the next modification was an addition of train collision detection. The most significant change was an update of the communication subsystem of TD. The subsystem included in the version 3.7 of TD allowed to control the simulation remotely by means of messages that emulate user interaction. For example, when a user clicked somewhere in the simulator, a message in the form `click x y`, where `x` and `y` are coordinates of the location where he or she clicked, has been sent. Events such as a train entering the layout or waiting for a green signal were not handled at all. The updated version communicates with TS2JC. It sends information about the status of signals and switches and about events triggered by train operation to TS2JC. From TS2JC, it receives new states of signals and switches, computed by the control module. The communication is described in more detail in Sect. 2.3. The last modification was an implementation of a scanning process that creates a list of track sections of the layout. By track sections we mean track segments between signals, switches and entry points. The entry points are places where a train may enter or leave the layout.



Entry	From	To	Exit	Train	Speed	Min.Del.	Min.Late	Status	Notes
4:10:00	e0	e1	4:12:30	Reg001	0	0	0	Wait...	
4:12:40	e1	e0	4:15:30	IC002	0	0	0	ready	

Fig. 1. Modified Train Director during simulation. Features specific to TD/TS2JC are marked with (red) dashed rectangles and labeled by letters C, D and L. (Color figure online)

New features have been also implemented to the GUI of the simulator (Fig. 1). These include the indication of the connection with TS2JC (C in Fig. 1) and presence of the control module (D) and the possibility to show labels (L) with the names of switches and signals in the layout. The scenario shown in Fig. 1 is probably the simplest one that is usable for teaching purposes. It is also utilized in the short course, presented in Sect. 4. The layout of the scenario contains one straight track (the thick black dashed line in Fig. 1) with two entry points (`e0`, `e1`) and two signals (`sig0`, `sig1`). As with the real railway, the signals guard the entrance to the track ahead of them. Here, `sig0` is meant for the trains coming from the west (from `e0`) and `sig1` for the trains coming from the east (from `e1`). In TD, the trains always obey the corresponding signals. The trains are represented by orange train engine icons. In Fig. 1, a train named Reg001 is passing the signal `sig0`, which is green. The scanning process mentioned above detects two sections in the layout. The first one is between `e0` and the signals and the second one is between the signals and `e1`. If there are more signals at the same place, the one guarding the section is used to name it. So, the sections in Fig. 1 are `(e0, sig1)` and `(sig0, e1)`.

Listing 1. Train schedule `route2sec.sch`.

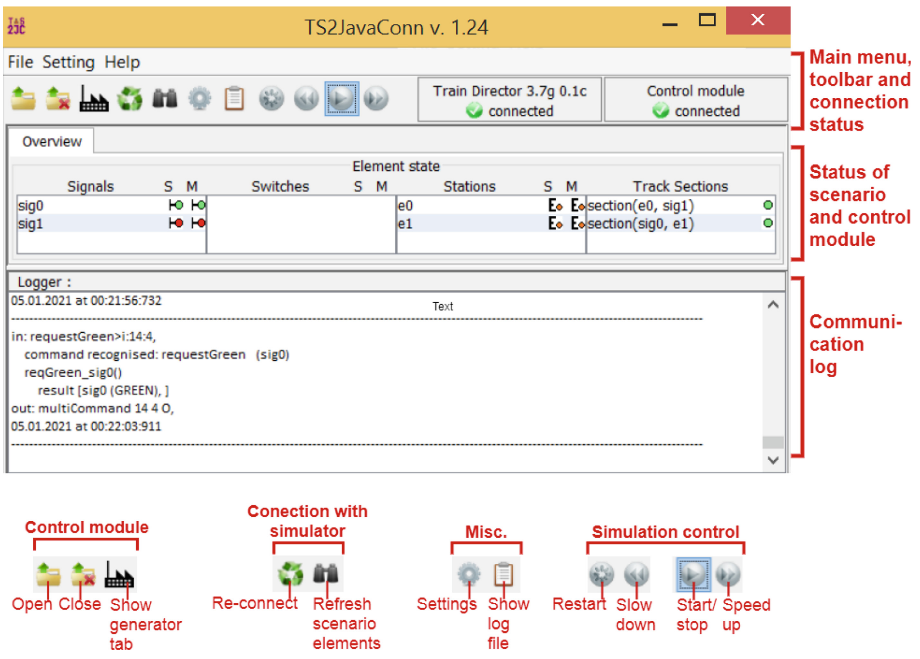
```
 1  #!trdir
 2  # no deadlock - delays between trains long enough
 3  Start: 4:10
 4  Train: Reg001
 5    Enter: 04:10, e0
 6           04:12:30, -, e1
 7  .
 8  Train: IC002
 9    Enter: 04:12:40, e1
10      04:15:30, -, e0
11  .
```

The trains operate according to a schedule, given in a form of a text file with the extension `sch`. The scenario in Fig. 1 uses the schedule shown in Listing 1. Line 1 is mandatory. Other lines starting with the "#" character are regarded as comments, such as line 2 with notes about the schedule. Line 3 defines the simulated time at the beginning of each simulation. The rest of the file contains train schedules. The first train (lines 4–7) is named Reg001 and it enters the layout from `e0` at 4:10. It should leave the layout through `e1` at 4:12:30. The second train is IC002. It travels in the opposite direction, entering the layout at 4:12:40. Its schedule is defined on lines 8–11. The end of each schedule is marked by a dot (lines 7 and 11).

2.2 TS2JavaConn

The second part of the toolset is a Java application called *TS2JavaConn*. Its name is a shortcut for Train Simulator to Java Connector. TS2JavaConn serves as a middleman between the modified Train Director and control modules.

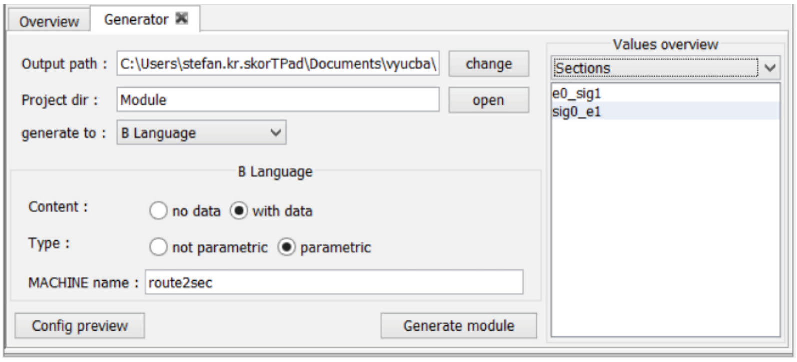
Control modules are loaded directly to TS2JavaConn, which uses the Java Reflection API to call its methods and process their return values. On the other hand, TS2JavaConn maintains a TCP connection with the updated communication subsystem of TD. The connection is used to receive messages about events occurring in TD during the simulation and to send commands that change the state of track devices in TD. Each message received from TD is translated to a call of a method from the control module. And each command sent to TD is constructed according to the state of the control module. TS2JavaConn allows various styles of control modules and each module has a configuration file that defines how messages from TD are translated to method calls and how to read the state of the control module. Concrete examples of control modules and configuration files are given in Sect. 3 and interaction between the module and TD is explained in detail in Sect. 2.3.



```
05.01.2021 at 00:21:56:732
--------------------------------------------------------------------------------
in: requestGreen>i:14:4,
  command recognised: requestGreen  (sig0)
  reqGreen_sig0()
    result [sig0 (GREEN), ]
out: multiCommand 14 4 0,
05.01.2021 at 00:22:03:911
--------------------------------------------------------------------------------
```

Fig. 2. Primary screen of TS2JavaConn (top) and description of its control panel buttons (bottom).

The primary screen of TS2JavaConn is shown in Fig. 2, which captures the application in the same moment as TD in Fig. 1. The screen is divided into three parts. The first one contains the main menu and toolbar for handling control modules, connection with the simulator and controlling the simulation remotely. It also includes connection status indicators with the same functionality as the ones added to TD (C and D in Fig. 1). The second part lists signals, switches, stations and track sections of the scenario. It also shows the state of these elements in TD (the "S" column) and in the control module (the "M" column). For the track sections, only the state from the control module is shown and it works only if the control module contains corresponding methods (getters). As we can see, the entry points are also treated as stations. The third part is a logger that shows detailed information about the communication between the simulator and the control module. In the case of an incorrect control module or configuration file, it also shows corresponding error messages. All the information shown in this part is saved to a log file, which can be opened from the toolbar. TS2JavaConn can be used with modified versions of two different simulators – Train Director and Open Rails [8]. When TS2JavaConn is opened, it searches for running instances of these simulators and let the user choose the one to connect to. The same thing happens after hitting the "Re-connect" button from the toolbar.



Output path : C:\Users\stefan.kr.skorTPad\Documents\vyucba\
Project dir : Module
generate to : B Language

B Language

Content : no data [✓ with data]
Type : not parametric [✓ parametric]
MACHINE name : route2sec

Values overview
Sections
e0_sig1
sig0_e1

Fig. 3. Control module generator screen of TS2JavaConn.

TS2JavaConn also offers a secondary screen with a control module generator (Fig. 3). The screen is activated by the "Show generator tab" button. The generator can create configuration files and template code for control modules in Java and languages of two formal methods – the B-Method and the Perfect Developer. The template code contains headers of all necessary methods (operations) and may also include variables representing the scenario elements. For Java and the B-Method, two types of control modules are available: parametric and non-parametric. Examples of both are given in Sect. 3.

2.3 Communication with Control Modules

As we will see later, in Sect. 3, each control module contains a *central class* with two types of methods:

– *getters*, which return values of module variables that correspond to states of track elements in the simulated scenario and
– *modifiers*, which are called when an event occurs in the simulated scenario. They may change the values of the module variables.

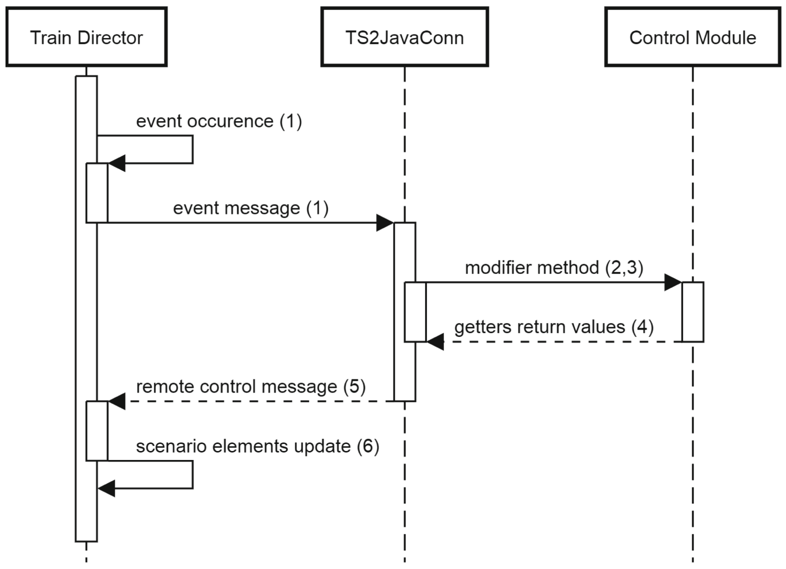


Fig. 4. UML sequence diagram illustrating event handling in TD/TS2JC. The numbers in brackets are numbers of steps in the description of the event handling process.

By *track elements* we mean entry points, signals, switches, stations and track sections. However, it is not necessary to define getters for all of them. In total, there are five types of events:

– A train requests to enter the scenario via an entry point.
– A train stops before a red signal and requests the signal to be cleared.
– A train departs from a station.
– A train leaves a track section.
– A train enters a track section.

TD/TS2JC handles every event in the following way (Fig. 4):

1. After an event occurs, TD composes a message about the event and sends it to TS2JavaConn. The message contains the type of the event and data about involved scenario elements and train.
2. TS2JavaConn reads the event message, received from TD, and identifies the corresponding modifier method of the control module.
3. TS2JavaConn calls the modifier method. Some parts of the method name or the values of its parameters may be composed from the data received from TD.
4. After the call of the modifier method is completed, TS2JavaConn calls all the getters of the control module and composes a remote control message from the values the getters return.
5. TS2JavaConn sends the remote control message to TD.

6. TD reads the message from TS2JavaConn and changes the states of the scenario elements accordingly.

The process of the event handling can be also observed in Fig. 1 and 2, which capture the simulation right after the event "Train `Reg001` requests `sig0` to be cleared" was handled. In the logger part of TS2JavaConn (Fig. 2), we can see that a modifier method[1] named `reqGreen_sig0` was called after the event. And finally, `sig0` has been changed to green in TD (Fig. 1).

Communication similar to the event handling also happens when a control module is opened: First, the control module is initiated by creating an instance of its central class. The constructor of the class sets the module variables to their initial values. TS2JavaConn reads these values by calling the getters. As in the case of the events, TS2JavaConn then composes a message from the values and sends the message to TD. Finally, TD sets the scenario elements accordingly and starts the simulation.

3 Control Modules and Configuration Files

When developing the TD/TS2JC toolset, one of the most important objectives was to be able to use outputs of verified software development tools as control modules without any or with minimal modifications. To reach this objective, it was necessary to support various forms of the control module interface, that is of the getters and the modifiers. The form of the interface is defined in a configuration file, accompanying each control module. In this section we present two distinct control modules for the scenario from Fig. 1. Both provide the same functionality but differ significantly in the interface.

The first one is introduced in Sect. 3.1 and is an example of so-called *non-parametric module*. This means that none of its getters and modifiers has input parameters and event data from TD are part of the names of the methods. The second one, in Sect. 3.2, is a fully *parametric module*, where all the event data translate to values of parameters of corresponding methods. It is also possible to create hybrid modules, where some of the data become parts of the method names and other are parameters.

The description of each control module and configuration file is given in the following way. First, the complete source code is presented as a listing. The source codes contain comments marking corresponding parts in terms introduced in Sect. 2.3. The comments start with "//" in the control modules and with "--" in the configuration files. Each listing is followed by a description giving more details about the code, referencing the corresponding code lines.

3.1 Non-parametric Module

From a conventional programmer point of view, it may look irrational to support non-parametric modules. This is because such modules require a separate

[1] The control module used in this case is the one from Listing 2 and the method is on lines 49–51.

method for each combination of the event and used data values received from TD. However, the support solves two problems related to the utilization of the toolset in teaching verified software development with formal methods.

First, languages of some formal methods (FMs) may not allow to use input parameters in specification units[2] that translate to the methods of the module. They may only allow units defining simple state transitions, without an external influence, which is usually represented by the input parameters.

Second, even if given formal method (FM) supports input parameters, the ability to use fully functional programs (control modules) without them may come very handy in the teaching process. For example, in an introductory part of a longer course. Or in a short course that teaches only the basics of the corresponding FM, where aspects of the method related to the utilization of input parameters are not tackled at all. The latter is also the case of the course introduced in Sect. 4, where a non-parametric one, similar to the module route2sec, presented here, is used. The complete source code of route2sec in Java can be found in Listing 2.

Listing 2. Non-parametric control module for the scenario from Fig. 1.

```java
public class route2sec {

//Sets defining states of track elements
//(entry points & signals, switches, sections)
  public enum ST_SIG {
    green(0), red(1);
    public final int index;
    ST_SIG(int index) { this.index = index; }
  }
  public enum ST_SWCH {
    switched(0), none(1);
    public final int index;
    ST_SWCH(int index) { this.index = index; }
  }
  public enum ST_SEC {
    free(0), occup(1);
    public final int index;
    ST_SEC(int index) { this.index = index; }
  }

//Variables for entry points, signals and sections
  private route2sec.ST_SIG e0,e1,sig0,sig1;
  private route2sec.ST_SEC e0_sig1,sig0_e1;

//Constructor setting the initial state of the elements
```

[2] We use the term "specification unit" as the corresponding parts (units) of formal specifications are named differently in different FMs. For example, they are called operations in the B-Method, events in the Event-B and schemas in the Perfect Developer.

```
  public route2sec() {
    e0 = ST_SIG.red; e1 = ST_SIG.red;
    sig0 = ST_SIG.red; sig1 = ST_SIG.red;
    e0_sig1 = ST_SEC.free; sig0_e1 = ST_SEC.free;
  }

//Getters for entry points and signals
  public route2sec.ST_SIG getEntry_e0() { return e0; }
  public route2sec.ST_SIG getEntry_e1() { return e1; }
  public route2sec.ST_SIG getSig_sig0() { return sig0; }
  public route2sec.ST_SIG getSig_sig1() { return sig1; }

//Modifier called when a train requests to enter from e0
  public void reqGreen_e0() {
    if (sig1 == ST_SIG.red && e0_sig1 == ST_SEC.free)
      e0 = ST_SIG.green; }

//Modifier called when a train requests to enter from e1
  public void reqGreen_e1() {
    if (sig0 == ST_SIG.red && sig0_e1 == ST_SEC.free)
      e1 = ST_SIG.green; }

//Modifier called when a train requests to clear sig0
  public void reqGreen_sig0() {
    if (e1 == ST_SIG.red && sig0_e1 == ST_SEC.free)
      sig0 = ST_SIG.green; }

//Modifier called when a train requests to clear sig1
  public void reqGreen_sig1() {
    if (e0 == ST_SIG.red && e0_sig1 == ST_SEC.free)
      sig1 = ST_SIG.green; }

//Modifiers called when a train enters the corresponding
//section from the corresponding direction
  public void enterNI_e0_sig1() {
    e0_sig1 = ST_SEC.occup;
    e0 = ST_SIG.red; sig1 = ST_SIG.red; }

  public void enterIN_sig0_e1() {
    sig0_e1 = ST_SEC.occup;
    sig0 = ST_SIG.red; e1 = ST_SIG.red; }

  public void enterNI_e1_sig0() {
    sig0_e1 = ST_SEC.occup;
    sig0 = ST_SIG.red; e1 = ST_SIG.red; }

  public void enterIN_sig1_e0() {
    e0_sig1 = ST_SEC.occup;
    e0 = ST_SIG.red; sig1 = ST_SIG.red; }

```

```
76 //Modifiers called when a train leaves the corresponding
77 //section from the corresponding direction
78   public void leaveNI_e0_sig1() { e0_sig1 = ST_SEC.free; }
79   public void leaveIN_sig1_e0() { e0_sig1 = ST_SEC.free; }
80
81   public void leaveIN_sig0_e1() { sig0_e1 = ST_SEC.free; }
82   public void leaveNI_e1_sig0() { sig0_e1 = ST_SEC.free; }
83
84 }
```

The whole module route2sec is defined in its central class, with the same name. It uses values from enumerated sets for the states of signals (the set ST_SIG), switches (ST_SWCH) and track sections (ST_SEC). The sets[3] are defined on lines 5–19 of the module. The set ST_SWCH (lines 10–14) can be excluded as the layout does not contain switches.

The instance variables, defined on lines 22–23, provide an internal representation of the state of the scenario. Here we have a separate variable for each entry point, signal and section and the variables have the same names as the corresponding elements in the scenario. However, such one-to-one correspondence between the elements and the variables is not mandatory. A programmer is free to choose whatever representation desired as the variables of the module are never accessed directly when communicating with TD. The variables are initialized in the constructor on lines 26–30.

Lines 33–36 contain the getters, returning the states of entry points and signals. These getters are mandatory[4]. The module may also include getters for the track sections, but their only purpose is to display states of the sections in TS2JavaConn.

Lines 39–56 define modifiers called when a train wants to enter the scenario from the corresponding entry point or clear the corresponding signal. All four of them work in the same way: "Check whether the section to be entered is free and closed from the other side. If yes, set the signals and entry points involved accordingly."

The next four modifiers (lines 60–74) respond to the "train entering a section" events. There are four of them, while the scenario contains only two sections. This is because there is a separate method for each direction. The direction to which the method belongs is defined by the order of the track element names in its header. For example, enterNI_e0_sig1 is called when a train enters the section `(sig0, e1)` from `e0` and enterIN_sig1_e0 when it enters the same section from `sig1`. The letter "N" in method names means "entry point" and "I" means "signal". These shortcuts have been introduced to ensure unambiguity when different types of track elements have the same names. Each of these methods marks the corresponding section as occupied and sets the signals guarding it to red. Similar modifiers for the "train leaving a section" events are defined on lines 78–82.

[3] Technically, the enumerated sets are classes, too.
[4] If a scenario contains switches, their getters are mandatory, too.

For TS2JavaConn to understand the control module from Listing 2, the configuration file shown in Listing 3 is needed.

Listing 3. Configuration file of the non-parametric module from Listing 2.

```
mainClassName=route2sec.class

-- Entry point representation and getters
entryState=ST_SIG
entryOpenState=green
entryCloseState=red
getEntryNames=getEntry_%name%
getEntryOut=%ST_SIG%

-- Signal representation and getters
sigState=ST_SIG
signalGreenState=green
signalRedState=red
getSignalNames=getSig_%name%
getSignalOut=%ST_SIG%

-- Switch representation and getters
swchState=ST_SWCH
switchOpenState=switched
switchCloseState=none
getSwitchNames=getSwch_%name%
getSwitchOut=%ST_SWCH%

-- Section representation
sectionState=ST_SEC
sectionFreeState=free
sectionOccupState=occup

-- Modifiers for train requests to enter the scenario,
-- clear a signal and depart a station
requestDepartureEntry=reqGreen_%name%
requestGreen=reqGreen_%name%
requestDepartureStation=%ignore%

-- Modifiers for section entering and leaving events
sectionEnter=
      enter%shortcutAct%%shortcutNxt%_%nameAct%_%nameNxt%
sectionLeave=
      leave%shortcutPre%%shortcutAct%_%namePre%_%nameAct%

-- Track element shortcuts
signalShortcut=I
switchShortcut=W
entryShortcut=N
```

The first line of the file in Listing 3 specifies the filename of the compiled version of the control module central class. Lines 4–27 define how the track elements, namely entry points, signals, switches and sections, are represented and how the corresponding getters look.

For entry points, the representation and getters definition is given on lines 4–8. In this case, the states of the entry points are values from an enumerated set named ST_SIG (line 4). It is also possible to use the integer (value %int%) or the boolean (value %boolean%) type for the state values. Line 5 defines the value for an opened entry point and line 6 for a closed entry point.

The interface of the entry point getters is given on lines 7–8. The name of the getter is specified on line 7 as a combination of a fixed part and the placeholder %name%, which means the corresponding entry point name in the scenario. Two additional placeholders can be used when naming the getters:

– %number% – the numerical part of the element name and
– %shortcut% – a shortcut of the corresponding track element type. The shortcuts are defined on lines 40–42.

It is possible to combine the placeholders. For example, if line 7 in Listing 3 has been defined as

```
7 getEntryNames=get%shortcut%%number%
```

then the getters for the entry points will be

```
public route2sec.ST_SIG getN0() { return e0; }
public route2sec.ST_SIG getN1() { return e1; }
```

The return type of the entry point getters is specified on line 8. If it is an enumerated set, as in this case, its name is enclosed in the percent signs (% ST_SIG%) and only the set already defined for the corresponding state values (line 4) can be used.

In the same way, the representation and getters are defined for signals (lines 11–15) and switches (lines 18–22). As this scenario does not contain switches, lines 18–22 can be omitted. We included them primarily to explain the switch state values, which may not be clear from the names of the corresponding properties. Line 19 defines the value used for a switch set to the diverging track (value switched) and line 20 the value for the straight track. The part for sections (lines 25–27) lacks the properties getSectionNames and getSectionOut as there are no section getters in the control module. These properties are used in the configuration file in Listing 5.

The names of the modifiers are given on lines 30–37. The names of the methods called when a train requests to enter the scenario (line 31) or to clear a signal (line 32) are defined in the same way as for the getters. Line 33 sets the names of modifiers called when a train leaves a station. Our module does not contain any stations. Therefore, we decided to use the %ignore% placeholder to indicate that the module does not handle such events. A rather complicated interface of the modifiers for the section entering (line 36) and section leaving (line 37) events requires six placeholders:

- %shortcutAct% and %nameAct% – the shortcut and name of the scenario element from which the train enters or leaves the track section,
- %shortcutNxt% and %nameNxt% – the shortcut and name of the element from which the train will leave the section it is now entering and
- %shortcutPre% and %namePre% – the shortcut and name of the element from which the train entered the section it is now leaving.

The configuration file ends with the definition of the shortcuts for signals, switches and entry points on lines 40–42. Line 41 is for switches, so it can be omitted.

3.2 Parametric Module

Albeit it is not so obvious in this case, parametric modules offer more compact interface as there is no need for a separate method for each combination of involved scenario elements. The parametric module route2secP (Listing 4) controls the scenario in the same way as the nonparametric route2sec from Listing 2, but there are several differences in the interface and representation of the scenario elements and their states:

- scenario elements are defined as members of the enumerated sets SIGNALS and SECTIONS,
- states of the scenario elements are expressed as integers,
- instance variables are arrays of the scenario elements state values and
- scenario elements related to the getters and modifiers are given as their parameters.

Listing 4. Parametric control module for the scenario from Fig. 1.

```java
public class route2secP {

//Sets defining track elements
//(entry points & signals, sections)
  public enum SIGNALS {
    e0(0), e1(1), sig0(2), sig1(3);
    public final int index;
    SIGNALS(int index) {
      this.index = index; }
  }
  public enum SECTIONS {
    e0_sig1(0), sig0_e1(1);
    public final int index;
    SECTIONS(int index) {
      this.index = index; }
  }

//Array variables for entry points & signals and sections
  private int[] signals = {0,0,0,0}; //entry p. & signals
  private int[] sections = {0,0};

//constructor (empty)
  public route2secP() {    }

//Getters for entry points & signals and sections
  public int getSig(route2secP.SIGNALS sig) {
    return signals[sig.index]; }
  public int getSec(route2secP.SECTIONS sec) {
    return sections[sec.index];}

//Modifier called when a train requests to enter from e0/e1
  public void reqEnter(route2secP.SIGNALS sig) {
    switch (sig) {
      case e0:
        if ((signals[SIGNALS.e0.index] == 0   &&
             signals[SIGNALS.sig1.index] == 0 &&
             sections[SECTIONS.e0_sig1.index] == 0)) {
                signals[SIGNALS.e0.index] = 1;
        } break;
      case e1:
        if ((signals[SIGNALS.e1.index] == 0   &&
             signals[SIGNALS.sig0.index] == 0 &&
             sections[SECTIONS.sig0_e1.index] == 0)) {
                signals[SIGNALS.e1.index] = 1;
        } break;
    }
  }

//Modifier called when a train requests to clear sig0/sig1
  public void reqGreen(route2secP.SIGNALS sig) {
    switch (sig) {
      case sig0:
        if ((signals[SIGNALS.sig0.index] == 0 &&
             signals[SIGNALS.e1.index] == 0   &&
             sections[SECTIONS.sig0_e1.index] == 0)) {
                signals[SIGNALS.sig0.index] = 1;
        } break;
      case sig1:
        if ((signals[SIGNALS.sig1.index] == 0 &&
             signals[SIGNALS.e0.index] == 0 &&
             sections[SECTIONS.e0_sig1.index] == 0)) {
                signals[SIGNALS.sig1.index] = 1;
        } break;
    }
  }

//Modifier called when a train enters a section
  public void enter(route2secP.SECTIONS sec) {
    switch (sec) {
```

```
70       case e0_sig1:
71         sections[SECTIONS.e0_sig1.index] = 1;
72         signals[SIGNALS.e0.index] = 0;
73         signals[SIGNALS.sig1.index] = 0;
74         break;
75       case sig0_e1:
76         sections[SECTIONS.sig0_e1.index] = 1;
77         signals[SIGNALS.e1.index] = 0;
78         signals[SIGNALS.sig0.index] = 0;
79         break;
80     }
81   }
82
83 //Modifier called when a train leaves a section
84   public void leave(route2secP.SECTIONS sec) {
85     sections[sec.index] = 0;  }
86 }
```

As in the case of route2sec, the parametric module is defined in one class (Listing 4). And again, the code of the class starts with enumerated sets declarations (lines 5–16 in Listing 4). However, the sets SIGNALS and SECTIONS hold scenario elements and not their states. These sets are needed because the elements are parameters of the methods of the module. On the other hand, the element states are integers (0 and 1) here, so no enumerated sets for them are necessary.

Regarding the enumerated sets, there is one more difference between this module and the non-parametric one. In Java, each member of an enumerated set is represented by its name and index. The non-parametric route2sec uses only the values while route2secP relies heavily on the indices. This is because the instance variables (lines 19–20) are arrays that hold values of the scenario elements states on the positions given by the corresponding indices in the enumerated sets (for signals in SIGNALS and for sections in SECTIONS). The variables are initialized when declared, so the constructor (line 23) is empty.

The getters are defined on lines 26–29 and the modifiers occupy the rest of the module. The getter getSig returns states of the entry points and signals and getSec of the track sections. A getter may have only one input parameter, the element which state it returns. The modifier reqEnter is called when a train wishes to enter the scenario via e0 or e1 and reqGreen when it requests to clear sig0 or sig1. All section entering events are handled by the method enter and all section leaving ones by leave. A parametric module may be defined in a different, probably simpler, way. The form presented here has been chosen because it is nearly identical to a parametric module when developed in the B-Method using the template code generated by TS2JavaConn.

The configuration file of the module route2secP can be found in Listing 5 and follows the same structure as the one in Listing 3. Regarding the differences, this file contains additional properties for method parameters and track element representation and the unnecessary properties related to switches are excluded.

A minor difference is also the utilization of the integer type (placeholder %int%) and the values 0 and 1 for the track element states.

Listing 5. Configuration file of the parametric module from Listing 4.

```
mainClassName=route2secP

-- Entry point representation and getters
entryState=%int%
entryOpenState=1
entryCloseState=0
entryIndex=SIGNALS
entryIndexName=%name%
getEntryNames=getSig
getEntryParams=%SIGNALS%
getEntryOut=%int%

-- Signal representation and getters
sigState=%int%
signalGreenState=1
signalRedState=0
sigIndex=SIGNALS
sigIndexName=%name%
getSignalNames=getSig
getSignalParams=%SIGNALS%
getSignalOut=%int%

-- Section representation and getters
sectionState=%int%
sectionFreeState=0
sectionOccupState=1
sectionIndex=SECTIONS
sectionIndexName=%west%_%east%
getSectionNames=getSec
getSectionParams=%SECTIONS%
getSectionOut=%int%

-- Modifiers for train requests to enter the scenario,
-- clear a signal and depart a station
requestDepartureEntry=reqEnter
requestDepartureEntryParams=%SIGNALS%
requestGreen=reqGreen
requestGreenParams=%SIGNALS%
requestDepartureStation=%ignore%

-- Modifiers for section entering and leaving events
sectionEnter=enter
sectionEnterParams=%SECTIONS%
sectionLeave=leave
sectionLeaveParams=%SECTIONS%
```

The properties defining the parameters of the getters and modifiers can be found on lines 10, 20, 30, 36, 38, 43 and 45 in Listing 5. In general, their values are comma-separated lists of placeholders, defining the types of the parameters of the corresponding methods. In this module, all methods have only one parameter, so there is always just one value for each property.

As track elements are now members of enumerated sets, additional properties are required to define them. For the entry points, these can be found on lines 7–8 in Listing 5. Line 7 specifies the name of the enumerated set and line 8 how the names of its members are constructed from the entry points in the scenario. The property on line 8 can use the placeholders %name%, %number% and %shortcut%, in the same way as already discussed in Sect. 3.1. The same properties for signals are on lines 17–18 and for sections on lines 27–28. The section names (line 28) are formed from

– the name of the track element on their west (left) end (placeholder %west%),
– the underscore and
– the name of the track element on their east (right) end (%east%).

How the section names look in the module route2secP can be seen on line 12 in Listing 4.

4 Teaching Verified Software Development in B-Method with TD/TS2JC Toolset

In this section, we describe a course on Software Development with the B-Method, which utilizes the toolset. The course is intended for events such as summer schools and its typical duration is four to six hours. The description provided here covers both the body of knowledge to be given to the course participants and the process of the course, including examples and tasks.

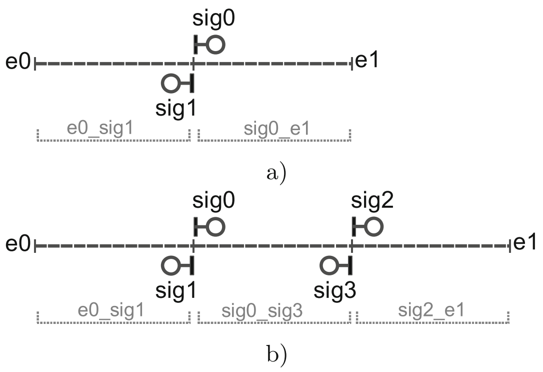


Fig. 5. Track layouts of scenarios used in the course: a straight track with two (a) and three (b) sections.

The language and development process of the B-Method is explained on a control module for a straight track with two sections (Fig. 5 a). The module is equivalent to the one in Listing 2. Within the course, the participants develop a similar control module for a straight track with three sections (Fig. 5 b). Both modules are non-parametric and, except of the class name on line 1, they use the same configuration file as the one in Listing 3.

The course starts with the lecturer informing the participants that the B-Method [1,2,11,15] was originally developed by J.R. Abrial and combines his previous invention, the Z-notation [16], with a minimalistic programming language, based on the language of Guarded commands [4] by E.W. Dijkstra. According to the taxonomy presented in [3], the B-Method belongs to so-called heavyweight formal methods as it involves theorem proving to verify software correctness.

4.1 Software Development Process of B-Method

The highlight of the method is the development process that fully incorporates formal verification. First, a formal specification of a system, consisting of components called abstract machines, is written. An abstract machine, or simply a machine, consists of a set of variables that defines its state and a set of operations that define state transitions. The specification of each machine (which has variables) contains a formula that defines its invariant properties. The B-Method allows to formally prove that these properties hold in every state of the machine. Machines are then developed to implementable components, called implementations.

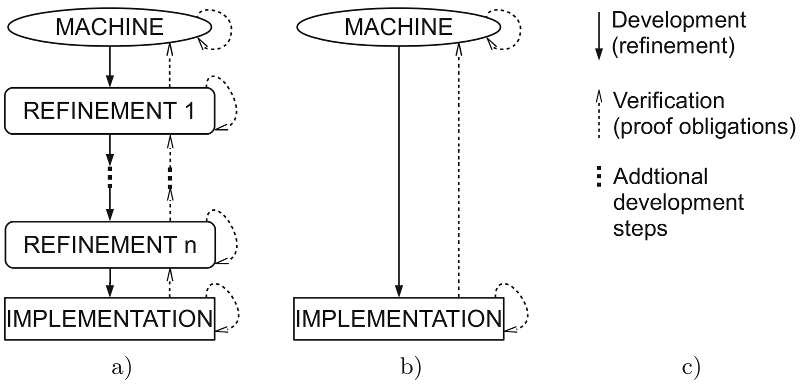


Fig. 6. Development of a specification component in B-Method: in general (a), in the course (b), legend (c).

This development process, which is also called stepwise refinement, consists of one or more steps. Multiple-steps process (Fig. 6 a) involves intermediate components, called refinements. One-step process (Fig. 6 b) goes directly from a machine to an implementation and it is the one chosen for this course. Refinements and implementations are components similar to machines. They contain invariant properties, too. These properties also define a relation between their variables and the variables of components they refine. And, again, it is possible to formally prove that they hold in each state of the component. In this way, it is possible to verify that the properties once defined at the abstract level (machines) still hold in the executable implementation. This is the reason why we can say that the B-Method offers a verified software development process. At each step, the specification may consist of multiple components and the number of components may vary. The number of refinement steps can also be different for each component. In this short course, we develop a control module that consists of one component, refined in one step from a machine to an implementation. Of course, the TD/TS2JC toolset allows for modules developed from multiple components, as it is shown in [9].

4.2 B-Language

The course continues with an explanation of the B-language, a specification language in which the components are written. The B-language can be divided to two parts:

– A *mathematical notation* to write expressions and predicates on data in terms of the Zermelo-Fraenkel set theory.
– The *Generalized Substitution Language* (*GSL*), a minimalistic programming language with the formal semantics defined by the weakest pre-condition calculus [4].

Table 1. Selected operators of the mathematical notation of B-language.

Operator	Meaning
&	and (logical conjunction)
not	not (logical negation)
=>	then (logical implication)
<=>	logical equivalence
=	equals
{	start of a set
}	end of a set
:	belongs to (a member to a set)

The mathematical notation is quite complex, fortunately we need just a small portion of it here. This portion is given in Table 1.

The commands of GSL are called *generalized substitutions* (*GS*) and those relevant to the course are listed in Table 2. The symbols introduced in Table 2 have the following meaning:

Table 2. Selected commands of GSL and their informal meaning.

Command	Meaning
skip	Do nothing
x := e	Assignment of values of expressions e to variables x
S1 ; S2	Sequential composition: do S1, then S2
S1 || S2	Parallel composition: do S1 and S2 at once
PRE E THEN S1 END	If E holds, do S1. Otherwise, do anything
SELECT E THEN S1 END	If E holds, do S1. Otherwise, do not execute
CHOICE S1 OR S2 END	Bounded choice: do S1 or S2
IF E THEN S1 ELSE S2 END	If E holds, do S1. Otherwise, do S2

– x is a comma-separated list of variables,
– e is a comma-separated list of expressions over variables, with the same length as x,
– S1 and S2 are GS (GSL commands) and
– P and E are predicates.

In the case that the ELSE branch is omitted in the IF command, S2 is considered equal to skip. There are two significant omissions in Table 2. The first one is so-called unbounded non-determinism, which is like the bounded choice, but allows to introduce local variables. The second one is a do-while loop, which includes a loop invariant. Both of them, together with other, derived, GSL commands, and the mathematical notation are described in [1,11,15].

The formal semantics of GS is defined in the *weakest pre-condition calculus* of E.W. Dijkstra [4]. The *weakest pre-condition of a GS S1 with respect to a post-condition P* is the predicate (1),

$$[\mathsf{S1}]\mathsf{P} \tag{1}$$

which is satisfied in exactly all states from which an execution of S1 is guaranteed to terminate in a state satisfying P.

The weakest pre-conditions of the commands from Table 2 can be found in Table 3. The operators are from the mathematical notation and are listed in Table 1. The notation (2)

$$\mathsf{P}[\mathsf{x} := \mathsf{e}] \tag{2}$$

is the predicate P with all free occurrences of variables from x replaced by the corresponding expressions from e.

There are two interesting things one may notice in Table 3. First, the IF command is just a combination of the commands CHOICE and SELECT and it can be written in the form (3).

$$\begin{array}{l}\mathsf{CHOICE\ SELECT\ E\ THEN\ S1\ END\ OR} \\ \qquad \mathsf{SELECT\ not(E)\ THEN\ S2\ END\ END}\end{array} \tag{3}$$

Table 3. Formal semantics of GSL commands from Table 2.

Command	Weakest pre-condition
[skip]P	P
[x := e]P	P[x := e]
[S1 ; S2]P	[S1]([S2]P)
[PRE E THEN S1 END]P	E & [S1]P
[SELECT E THEN S1 END]P	E =>[S1]P
[CHOICE S1 OR S2 END]P	[S1]P & [S2]P
[IF E THEN S1 ELSE S2 END]P	(E =>[S1]P) & (not(E) =>[S2]P)

Second, the semantics of the parallel composition is not defined here. This is because the simplest case (4) of the parallel composition can be written in the form (5).

```
x1 := e1 || x2 := e2        (4)
x1,x2 := e1,e2              (5)
```

The B-Method also offers rules to transform more complicated cases of multiple GS to the case (4). These rules are not needed in the course, but an interested reader can find them in [1,11].

Within the course, the comprehension of this theory can be fortified by Exercise 1.

Exercise 1. Generalized Substitution Syntax and Semantics.
Task
Compute the weakest pre-condition (6).

```
[IF sig1=red & e0_sig1=free
 THEN e0:=green || e0_sig1:=occup END] (e0=green)        (6)
```

Solution
First, we use the semantics of IF from Table 3 and the form (5) of (4) to rewrite (6) to (7).

```
  ( (sig1=red & e0_sig1=free) => [e0,e0_sig1:=green,occup](e0=green) )
& ( not(sig1=red & e0_sig1=free) => [skip](e0=green) )        (7)
```

Applying the semantics of skip and := from Table 3 to (7), we get (8).

```
  ( (sig1=red & e0_sig1=free) => (green=green) )
& ( not(sig1=red & e0_sig1=free) => (e0=green) )        (8)
```

The form (8) is equivalent to (9).

```
  ( (sig1=red & e0_sig1=free) => true )
& ( not(sig1=red & e0_sig1=free) => (e0=green) )        (9)
```

According to the definition of the logical conjunction and implication, (9) can be further reduced to (10), and, finally, to (11), which is the final form of (6).

$$\mathsf{true}\ \&\ (\mathsf{not}(\mathsf{sig1{=}red}\ \&\ \mathsf{e0\_sig1{=}free}) \Rightarrow (\mathsf{e0{=}green})) \tag{10}$$

$$\mathsf{not}(\mathsf{sig1{=}red}\ \&\ \mathsf{e0\_sig1{=}free}) \Rightarrow (\mathsf{e0{=}green}) \tag{11}$$

(End of Exercise 1)
■

4.3 Abstract Specification

A development of a software system in the B-Method starts with a formal abstract specification, consisting of (abstract) machines. A typical machine resembles an object in the object oriented programming as it encapsulates a set of variables, defining its state, with a set of operations, defining state transitions.

A machine is defined in a textual form consisting of several clauses. Only one of them, the MACHINE clause, which defines its name and may also list its formal parameters, is mandatory. To cover all purposes a machine can serve and corresponding combinations of clauses is out of the scope of this short course. Therefore, we will limit ourselves to the clauses we need for the control modules to be developed. And we explain them on a particular example of a machine representing a control module for the scenario from Fig. 5 a). But before that, in Exercise 2, we use the Train Director part of the TD/TS2JC toolset to emulate a process of customer requirements analysis, which should result in the invariant properties of the machine.

Exercise 2. From requirements to invariant properties.

Task

1. Launch the version of Train Director that is a part of the TD/TS2JC toolset and open the railway scenario `route2sec.trk`, with the track layout as in Fig. 5 a), in it.
2. Imagine that your task is to develop a control module for this scenario. The control module
   – represents entry points and signals by variables e0, e1 and sig0, sig1 with values green and red,
   – represents track sections by variables e0_sig1, sig0_e1 with values free and occup (occupied),
   – reacts to a request from a train to enter a section by setting the corresponding signal or entry point and
   – assumes that all trains obey the values it sets for the entry points and signals (i.e. a train enters a section only when the corresponding signal (entry point) is green).
3. Specify invariant properties that ensure safety of the trains in the scenario

- informally, in English and
- formally, using the mathematical notation of the B-language and the variables defined above.

Use simulation of the scenario in the Train Director to explore possible situations. You can clear the signals manually by clicking on them and change the train schedule by editing the text file `route2sec.sch`.

Solution
Informally, the invariant properties can be specified as follows:

1. Only one of the signals (entry points) guarding a section can be green.
2. If any of the signals (entry points) guarding a section is green, the section itself must be free.

These statements can be formally expressed in several ways. One of them is given in Listing 6 on lines 12–13 (the first statement) and lines 14–16 (the second statement).

(End of Exercise 2)
■

The machine specifying a non-parametric controller for the scenario from Fig. 5 a) can be found in Listing 6. The interface and functionality of the controller is identical to the Java version from Listing 2 and its final, executable, version will use the same configuration file (Listing 3).

Listing 6. Machine route2sec of a non-parametric module for the two section track from Fig. 5 a).

```
 1 MACHINE route2sec
 2 SETS
 3    ST_SIG={green, red};
 4    ST_SWCH={switched, none};
 5    ST_SEC={free,occup}
 6
 7 CONCRETE_VARIABLES e0, e1, sig0, sig1, e0_sig1, sig0_e1
 8
 9 INVARIANT
10    e0:ST_SIG & e1:ST_SIG & sig0:ST_SIG & sig1:ST_SIG &
11    e0_sig1:ST_SEC & sig0_e1:ST_SEC &
12    (e0=green => sig1=red) & (sig1=green => e0=red)  &
13    (e1=green => sig0=red) & (sig0=green => e1=red)  &
14    (e0=green => e0_sig1=free) & (sig1=green => e0_sig1=free)
15    &
16    (e1=green => sig0_e1=free) & (sig0=green => sig0_e1=free)
17
18 INITIALISATION
19    e0:=red || e1:=red || sig0:=red || sig1:=red ||
20    e0_sig1:=free || sig0_e1:=free
```

```
21
22 OPERATIONS
23   ss <-- getSig_sig0 = BEGIN ss:=sig0 END;
24   ss <-- getSig_sig1 = BEGIN ss:=sig1 END;
25   ss <-- getEntry_e0 = BEGIN ss:=e0 END;
26   ss <-- getEntry_e1 = BEGIN ss:=e1 END;
27
28   reqGreen_e0 =
29     IF sig1=red & e0_sig1=free THEN e0:=green END;
30   reqGreen_e1 =
31     IF sig0=red & sig0_e1=free THEN e1:=green END;
32   reqGreen_sig0 =
33     IF e1=red & sig0_e1=free THEN sig0:=green END;
34   reqGreen_sig1 =
35     IF e0=red & e0_sig1=free THEN sig1:=green END;
36
37   enterNI_e0_sig1 =
38     BEGIN e0_sig1:=occup || e0:=red || sig1:=red END;
39   enterIN_sig0_e1 =
40     BEGIN sig0_e1:=occup || sig0:=red || e1:=red END;
41   enterNI_e1_sig0 =
42     BEGIN sig0_e1:=occup || sig0:=red || e1:=red END;
43   enterIN_sig1_e0 =
44     BEGIN e0_sig1:=occup || e0:=red || sig1:=red END;
45
46   leaveNI_e0_sig1 = BEGIN e0_sig1:=free  END;
47   leaveIN_sig0_e1 = BEGIN sig0_e1:=free  END;
48   leaveNI_e1_sig0 = BEGIN sig0_e1:=free  END;
49   leaveIN_sig1_e0 = BEGIN e0_sig1:=free  END
50
51 END
```

The MACHINE clause with the machine name (route2sec, on line 1 in Listing 6) is followed by the SETS clause on lines 2 to 5. This clause defines three enumerated sets with their members in curly brackets. They are considered types in the B-language. Line 4 can be omitted as the scenario does not contain switches.

The CONCRETE_VARIABLES clause names state variables of the machine. A machine may have two types of state variables. The first one is concrete variables, as in this case. Such variables remain the same in each subsequent refinement or implementation of the component. Therefore, there are certain restrictions on them as they must be implementable, that is automatically translatable to a common programming language. For the second type, we have the ABSTRACT_VARIABLES clause and these variables can be of any type definable in the B-language.

The invariant properties of the machine are specified as a predicate in the INVARIANT clause (lines 9–16). It is divided into the typing invariant (lines 10–11), defining the types of the state variables, and safety properties (lines 12–16) that are those formulated in Exercise 2.

The next clause is INITIALISATION (lines 18–20) with a command in GSL, which assigns initial values to all state variables. It regards all sections as empty and sets all signals and entry points to red.

GSL is also used in the OPERATIONS clause (lines 22–51) with all the operations of the component. The operations are separated by semicolons and their interface and functionality is the same as that of the methods in the Java module in Listing 2. In the B-language, a general form of an operation is (12),

$$\begin{array}{c}\mathsf{y} \mathrel{<\!\!--} \mathsf{op(x)} = \\ \mathsf{PRE\ P\ THEN\ S\ END}\end{array} \tag{12}$$

where y is a comma-separated list of its output parameters, op its name and x a comma-separated list of its input parameters. The predicate P, called the precondition of the operation, defines conditions under which it should be called. In operations with input parameters, it also defines their properties, including types. S is a command (a GS) that forms the body of the operation. For machines, it is required that operations are atomic state transitions without intermediate states. Because of this, they cannot contain the sequential compositions or loops.

If P is true and there are no input parameters, the form (12) is reduced to (13). This is the case of getters in our machine (lines 23–26). Remaining operations do not even have output parameters so they are written in the form (14). If S contains only compositions and assignments, it is common to place it between the keywords BEGIN and END. All operations in Listing 6, except of those in lines 28–35, use these keywords.

$$\mathsf{y} \mathrel{<\!\!--} \mathsf{op} = \mathsf{S} \tag{13}$$

$$\mathsf{op} = \mathsf{S} \tag{14}$$

Verification of Machine. To verify the correctness of a formal specification written in the B-language, one must prove a set of formulas, called proof obligations (PObs), for each machine of the specification. To explain this topic in a concise way, we restrict ourselves to machines like the one in Listing 6. In general, such a machine can be written as in Listing 7.

Listing 7. General form of a machine with clauses as in Listing 6.

```
MACHINE M
SETS St
CONCRETE_VARIABLES v
INVARIANT I
INITIALISATION T
OPERATIONS
   y <-- op(x) =
      PRE P THEN S END
END
```

The PObs for the machine are (15) and (16). The POb (16) must be proved for every operation of the machine.

$$[\mathsf{T}]\mathsf{I} \tag{15}$$
$$\mathsf{P}\ \&\ \mathsf{I} => [\mathsf{S}]\mathsf{I} \tag{16}$$
$$\mathsf{I} => [\mathsf{S}]\mathsf{I} \tag{17}$$

The POb (15) means that the initialisation must establish the invariant and (16) that each operation must preserve it. If P is true, (16) is reduced to (17).

Exercise 3. Proving the proof obligations.
Task
For the machine route2sec from Listing 6, prove (17) for the operation req-Green_e0.

Solution
The POb has the form (18). The letter I represents the invariant of route2sec, that is lines 10–16 from Listing 6.

$$\mathsf{I => [IF\ sig1{=}red\ \&\ e0\_sig1{=}free\ THEN\ e0{:=}green\ END]I} \tag{18}$$

After applying the GS semantics (Table 3) to (18), we get (19).

$$\begin{gathered}\mathsf{I => (\ ((sig1{=}red\ \&\ e0\_sig1{=}free)\ {=>}[e0{:=}green]I)\ \&} \\ \mathsf{(not(sig1{=}red\ \&\ e0\_sig1{=}free)\ {=>}I))\ )}\end{gathered} \tag{19}$$

In the rest of the exercise, we use the tautologies (20)–(22) of the propositional logic.

$$\mathsf{(\ a => (b\ \&\ c)) <=> ((a => b)\ \&\ (a => c))} \tag{20}$$
$$\mathsf{(a => (b => c)) <=> ((\ a\ \&\ b) => c)} \tag{21}$$
$$\mathsf{(a\ \&\ b)\ \ =>\ \ b} \tag{22}$$

Utilizing (20), we can split (19) to (23) and (24).

$$\mathsf{I => ((sig1{=}red\ \&\ e0\_sig1{=}free) => [e0{:=}green]I)} \tag{23}$$
$$\mathsf{I => (not(sig1{=}red\ \&\ e0\_sig1{=}free) => I)} \tag{24}$$

Considering (21), the formulas (23) and (24) can be rewritten to (25) and (26).

$$\mathsf{(I\ \&\ (sig1{=}red\ \&\ e0\_sig1{=}free)) => [e0{:=}green]I} \tag{25}$$
$$\mathsf{(I\ \&\ not(sig1{=}red\ \&\ e0\_sig1{=}free)) => I} \tag{26}$$

According to (22), (26) is true. What remains is to resolve (25). This requires to "dive into" the invariant I of route2sec, which is quite a long formula, consisting of 14 conjuncts. Therefore, in the rest of this solution and starting with (27), we omit those conjuncts that repeat in the same form on both sides of the implication and are not important for the proof.

$$\begin{aligned}&\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}}\\&\quad\mathsf{[e0{:=}green]\ (e0{:}ST\_SIG\ \&\ (e0{=}green\ {=}{>}\ sig1{=}red)\ \&}\\&\qquad\qquad\mathsf{(sig1{=}green\ {=}{>}\ e0{=}red)\ \&}\\&\qquad\qquad\mathsf{(e0{=}green\ {=}{>}\ e0\_sig1{=}free))}\end{aligned}\tag{27}$$

First, we use the semantics of assignment (Table 3) to transform (27) to (28).

$$\begin{aligned}&\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}}\\&\quad\mathsf{(green{:}ST\_SIG\ \&\ (green{=}green\ {=}{>}\ sig1{=}red)\ \&}\\&\quad\mathsf{(sig1{=}green\ {=}{>}\ green{=}red)\ \&}\\&\quad\mathsf{(green{=}green\ {=}{>}\ e0\_sig1{=}free))}\end{aligned}\tag{28}$$

Some of the expressions in (28) can be reduced to true or false, resulting in (29).

$$\begin{aligned}&\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}}\\&\quad\mathsf{(true\ \&\ (true\ {=}{>}\ sig1{=}red)\ \&}\\&\quad\mathsf{(sig1{=}green\ {=}{>}\ false)\ \&}\\&\quad\mathsf{(true\ {=}{>}\ e0\_sig1{=}free))}\end{aligned}\tag{29}$$

Considering the definition of logical implication and conjunction, (29) can be further reduced to (30).

$$\begin{aligned}&\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}}\\&\quad\mathsf{(sig1{=}red\ \&}\\&\quad\mathsf{not(sig1{=}green)\ \&}\\&\quad\mathsf{e0\_sig1{=}free)}\end{aligned}\tag{30}$$

According to (20), (30) can be split into 3 separate formulas, (31)–(33), to prove.

$$\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}\ sig1{=}red}\tag{31}$$

$$\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}\ not(sig1{=}green)}\tag{32}$$

$$\mathsf{(sig1{:}ST\_SIG\ \&\ sig1{=}red\ \&\ e0\_sig1{=}free)\ {=}{>}\ e0\_sig1{=}free}\tag{33}$$

Utilizing (22), (31) and (33) can be reduced to true directly as the right-hand side of the implication is one of the conjuncts on the left-hand side in both cases. And because the set ST_SIG consists of only two members, red and green, the right-hand side of (32) follows from the first two conjuncts on the left-hand side.

(End of Exercise 3)
■

B-Method in Atelier B. After getting familiar with the B-language and abstract machines and trying the formal verification in a pen-and-paper way, it is time to get some experience with Atelier B, the development environment and prover for the B-Method. In Exercise 4, we create a new project in Atelier B with the machine from Listing 6, which we type check and prove. This and the following exercises use the archive [7] of support materials and assume that the archive is unpacked into the folder `C:/VSD`. If one chooses another folder, he or she has to alter the steps accordingly.

Exercise 4. Machine specification and proof in Atelier B.
Instructions

1. Unpack the course package [7] to `C:/VSD`.
2. If not yet installed, download Atelier B from [18], install and run it. The *primary window* of Atelier B appears. The following steps are carried out in Atelier B.
3. Create a new workspace
   – workspace name: `bcourse`.
   – workspace database directory: `C:/VSD/bdb`.
4. Set the “Default project directory” to `C:/VSD/Bprojects`.
5. Create a new project called `route2sec`.
6. In the left panel (“Workspaces”), right click on the name of the project and choose “Add Components”.
7. In the dialog “Select one or more files to add”, locate and open the file `C:/VSD/Bprojects/route2sec/route2sec.mch`. It contains the machine from Listing 6.
8. In the *main part of the primary window*, which is located right to the “Workspaces” panel, choose “Classical view” from the dropdown menu (if not already chosen). A list of project components appears in the main part. The list contains `route2sec.mch` only.
9. Double click on the `route2sec.mch` in the list. This opens the *editor window* of Atelier B.
10. Explore the possibilities of the editor window and close it without saving changes in the file.
11. Right click on the `route2sec.mch` in the main part of the primary window and choose “Type check”. This will check the syntax of the component. Provided that you didn’t change anything in the file, this task should finish with success.
12. In the same way as in the previous step, choose “Generate PObs”. This will generate the proof obligations of the component. There should be eight of them.
13. In the same way as in the previous step, choose “Proof” and then “Automatic (Force 0)”. This will launch the *automatic prover* of Atelier B, which tries to prove the generated PObs of the component. The prover can be launched with different amount of resources (memory and time) allocated. There are four options in the menu - from the least amount (“Force 0”) to the greatest amount (“Force 3”). As the PObs of `route2sec.mch` are simple, Force 0 is sufficient.

14. Choose the "Proof" option, as in the previous step, and then "Interactive Proof". This opens the *interactive prover window* of Atelier B. The interactive prover is used for human-assisted proving when the automatic prover fails.
15. In the part "Situation" of the interactive prover window, double click on `route2sec` and then on `PO0`. The formula of the corresponding POb appears in the main part of the prover window. Notice the similarity between this formula and the ones we had in Exercise 3.
16. Close the interactive prover window and return to the primary window.
17. Notice that the type checking, POb generation and some of the proof options are also available from the toolbar.
18. If time allows, explore the functionality of Atelier B further. Corresponding documentation can be found in the main menu ("Help" and then "Manuals").

(End of Exercise 4)
■

4.4 Refinement to Implementation

The B-Method allows for a sophisticated development (refinement) process, with multiple steps and changes in both data representation and functionality of operations. Considering the limited duration of the course, we opted for a minimalistic form of refinement. This consists of only one step, directly from a machine to an implementation. And this step is only necessary because of certain limitations of different types of components in the B-Method. As we mentioned earlier, machine operations must be atomic state transitions, so sequential composition and loops are prohibited. In implementations, only commands compatible with those of sequential imperative programming languages are allowed. This rules out parallel composition and PRE, SELECT and CHOICE in their pure form (IF is allowed). Abstract (unimplementable) variables and constants are forbidden, too. On the other hand, sequential composition and loops can be used. Refinements, as intermediate components, are a mixed bag. They can use both the abstract and concrete data and all commands, except of the loops, are allowed in them. All refinements and the implementation of a machine must have the same interface as the machine. By the interface we mean the list of component parameters and headers of its operations. No operation can be added or removed during the refinement process.

A straightforward implementation of the machine route2sec is the component route2sec_i in Listing 8.

Listing 8. Implementation component, refined from the abstract machine in Listing 6.

```
IMPLEMENTATION route2sec_i
REFINES route2sec

INITIALISATION
  e0:=red; e1:=red; sig0:=red; sig1:=red;
```

```
 6     e0_sig1:= free ; sig0_e1:= free
 7
 8   OPERATIONS
 9     ss <-- getSig_sig0 = BEGIN ss:=sig0 END;
10     ss <-- getSig_sig1 = BEGIN ss:=sig1 END;
11     ss <-- getEntry_e0 = BEGIN ss:=e0 END;
12     ss <-- getEntry_e1 = BEGIN ss:=e1 END;
13
14     reqGreen_e0 =
15       IF sig1=red & e0_sig1=free THEN e0:=green END;
16     reqGreen_e1 =
17       IF sig0=red & sig0_e1=free THEN e1:=green END;
18     reqGreen_sig0 =
19       IF e1=red & sig0_e1=free THEN sig0:=green END;
20     reqGreen_sig1 =
21       IF e0=red & e0_sig1=free THEN sig1:=green END;
22
23     enterNI_e0_sig1 =
24       BEGIN e0_sig1:=occup ; e0:=red ; sig1:=red END;
25     enterIN_sig0_e1 =
26       BEGIN sig0_e1:=occup ; sig0:=red ; e1:=red END;
27     enterNI_e1_sig0 =
28       BEGIN sig0_e1:=occup ; sig0:=red ; e1:=red END;
29     enterIN_sig1_e0 =
30       BEGIN e0_sig1:=occup ; e0:=red ; sig1:=red END;
31
32     leaveNI_e0_sig1 = BEGIN e0_sig1:=free  END;
33     leaveIN_sig0_e1 = BEGIN sig0_e1:=free  END;
34     leaveNI_e1_sig0 = BEGIN sig0_e1:=free  END;
35     leaveIN_sig1_e0 = BEGIN e0_sig1:=free  END
36
37   END
```

There are several differences between the components route2sec and route2sec_i:

– The keyword MACHINE is replaced by IMLEMENTATION (line 1 in Listing 8).
– The clauses SETS and CONCRETE_VARIABLES are not present as the ones already defined in the machine are sufficient.
– The clause INVARIANT is omitted as no new variables are introduced and there is no need to define new properties over the concrete variables from the machine.
– All parallel compositions are replaced by sequential compositions in the INITIALISATION and OPERATIONS clauses.

Verification of Implementation. Refinements and implementations are verified against themselves and components they refine. Again, we will not deal with the most general case but only with a simplified one, as given in Listing 9.

Listing 9. General form of an implementation with clauses as in Listing 8 and INVARIANT.

```
IMPLEMENTATION M_i
REFINES M
INVARIANT J
INITIALISATION T1
OPERATIONS
   y <-- op(x) =
      BEGIN S1 END
END
```

The PObs for the implementation are (34) and (35) and (35) must be proved for every operation of the implementation. The PObs of the refinement components have the same form.

$$\text{[T1](not([T] not(J)))} \tag{34}$$
$$\text{P \& I \& J => [S1'](not([S] not(J \& y'=y)))} \tag{35}$$

While these PObs look rather complicated, their resolution is trivial in the case of route2sec_i. This is because

- the implementation route2sec_i does not contain invariant, so J is true and
- the weakest preconditions of the operation bodies of the machine (S) and the implementation machine (S1) are the same, because the right-hand sides of the assignments do not contain any state variables (that occur on the left-hand sides).

When looking on the general form of machine operation (12), and POb (35), one may wonder why we did not use PRE instead of IF in the machine route2sec (Listing 6). It is true that if we replace all IF keywords with PRE in Listing 6, then such machine can be refined to the implementation from Listing 8 and the verification will go without problems. But, it can also be refined to an implementation that differs from the one in Listing 8 in additional ELSE parts of the IF commands. And it will verify perfectly fine, regardless on what is inside the ELSE parts. In the case that these operations are called from another component in the same specification, it is OK. Because, in such a case it will not be possible to prove any component that calls them outside of their pre-conditions. But the operations of our verified control module will be called from outside of the verified part, where no one cares whether any conditions are met.

Implementation in Atelier B and BKPI Compiler. What remains is to finish the development of our control module in Atelier B and translate it to an executable form that can be run with the TD/TS2JC toolset. This is done in Exercise 5. The exercise requires both Java Runtime Environment and Java Development Kit installed. Similarly to Exercise 4, it assumes that the archive [7] is unpacked to `C:/VSD`.

Exercise 5. Implementation to executable code in Atelier B and BKPI compiler.
Instructions

1. In Atelier B, open the project `route2sec`, created in Exercise 4.
2. In the left panel ("Workspaces"), right click on the name of the project and choose "Add Components".
3. In the dialog "Select one or more files to add", locate and open the file `C:/VSD/Bprojects/route2sec/route2sec_i.imp`. It contains the implementation from Listing 8.
4. Type check, generate and prove PObs of `route2sec_i.imp` in the same way as for `route2sec.mch` in Exercise 4.
5. Close Atelier B.
6. Run the BKPI compiler.
   It is the file `C:/VSD/BKPICompiler/BKPIcompiler.jar` and we will use it to translate `route2sec_i.imp` to Java.
7. In the the BKPI compiler, right click on `route2sec` and choose "Generate Java code".
8. In a file manager (e.g. Explorer), navigate to the folder `C:/VSD/Bprojects/route2sec/java`.
9. Delete `MainClass.java` and compile `route2sec.java`. To compile, just run `compile.bat`.
10. Run Train Director and open the file `C:/VSD/scenarios/route2sec.trk` in it.
11. Run TS2JavaConn and load (open) the module `C:/VSD/Bprojects/route2sec/java/route2sec.class` in it.
12. Start the simulation, in Train Director or TS2JavaConn.

(End of Exercise 5)
■

4.5 Three Sections Control Module Development Project

Finally, the course participants may try the verified development process on a control module for the three track sections scenario from Fig. 5 b).

Exercise 6. Development of a control module for the three sections scenario.
Task
Develop a verified control module for the three track sections scenario from Fig. 5 b). The invariant of the machine of the module has to contain safety conditions that prevent collision of trains in the scenario. Use the control module template generator of TS2JavaConn to create an initial form of the module machine in the B-language. Alternatively, you can start with the machine from Listing 10. Follow the TODO comments to modify the machine.

Listing 10. Initial form of the abstract machine route3sec.

```
MACHINE route3sec
SETS
```

```
  ST_SIG={green, red};
  ST_SWCH={switched, none};
  ST_SEC={free,occup}

CONCRETE_VARIABLES
  e0, e1, sig0, sig1, sig2, sig3,
  e0_sig1, sig0_sig3, sig2_e1

INVARIANT
  e0:ST_SIG & e1:ST_SIG &
  sig0:ST_SIG & sig1:ST_SIG & sig2:ST_SIG & sig3:ST_SIG &
  e0_sig1:ST_SEC & sig0_sig3:ST_SEC & sig2_e1:ST_SEC &
  (e0=green => sig1=red) & (sig1=green => e0=red)
  /*TODO: finish safety conditions for the relations
          between signals (entry points)*/
  &
  (e0=green => e0_sig1=free) &
  (sig1=green => e0_sig1=free)
  /*TODO: finish safety conditions for the relations
          between signals (entry points) and sections*/

INITIALISATION
  e0:=red || e1:=red ||
  sig0:=red || sig1:=red || sig2:=red || sig3:=red ||
  e0_sig1:= free || sig0_sig3:= free || sig2_e1:= free

OPERATIONS
  ss <-- getSig_sig0 = BEGIN ss:=sig0 END;
  ss <-- getSig_sig1 = BEGIN ss:=sig1 END;
  ss <-- getSig_sig2 = BEGIN ss:=sig2 END;
  ss <-- getSig_sig3 = BEGIN ss:=sig3 END;
  ss <-- getEntry_e0 = BEGIN ss:=e0 END;
  ss <-- getEntry_e1 = BEGIN ss:=e1 END;

  reqGreen_e0 =
    IF sig1=red & e0_sig1=free THEN e0:=green END;
  reqGreen_e1   = skip;
  reqGreen_sig0 = skip;
  reqGreen_sig1 = skip;
  reqGreen_sig2 = skip;
  reqGreen_sig3 = skip;
  /*TODO: replace skip in the previous operation
          with appropriate commands*/

  enterNI_e0_sig1 =
    BEGIN e0_sig1:=occup || e0:=red || sig1:=red END;
  enterII_sig0_sig3 =
    BEGIN sig0_sig3:=occup || sig0:=red || sig3:=red END;
  enterIN_sig2_e1 =
    BEGIN sig2_e1:=occup || sig2:=red || e1:=red   END;

  enterNI_e1_sig2 =
    BEGIN sig2_e1:=occup || sig2:=red || e1:=red   END;
  enterII_sig3_sig0 =
    BEGIN sig0_sig3:=occup || sig0:=red || sig3:=red END;
  enterIN_sig1_e0 =
    BEGIN e0_sig1:=occup || e0:=red   || sig1:=red END;

  leaveNI_e0_sig1   = BEGIN e0_sig1:=free   END;
  leaveII_sig0_sig3 = BEGIN sig0_sig3:=free END;
  leaveIN_sig2_e1   = BEGIN sig2_e1:=free   END;
  leaveNI_e1_sig2   = BEGIN sig2_e1:=free   END;
  leaveII_sig3_sig0 = BEGIN sig0_sig3:=free END;
  leaveIN_sig1_e0   = BEGIN e0_sig1:=free   END

END
```

Solution
In essence, the process of the control module development is the same as in Exercise 4 and Exercise 5. The parts that must be changed in the machine from Listing 10 can be found in Listing 11. The differences between the machine and its implementation are the same as between the ones in the aforementioned exercises.

Listing 11. Invariant and selected operations of the final form of the machine route3sec from Listing 10.

```
INVARIANT
  e0:ST_SIG & e1:ST_SIG &
  sig0:ST_SIG & sig1:ST_SIG & sig2:ST_SIG & sig3:ST_SIG &
  e0_sig1:ST_SEC & sig0_sig3:ST_SEC & sig2_e1:ST_SEC &
  (e0=green => sig1=red) & (sig1=green => e0=red) &
  (sig0=green => sig3=red) & (sig3=green => sig0=red) &
  (e1=green => sig2=red) & (sig2=green => e1=red) &
  (e0=green   => e0_sig1=free)   &
  (sig1=green => e0_sig1=free)   &
  (sig0=green => sig0_sig3=free) &
  (sig3=green => sig0_sig3=free) &
  (e1=green   => sig2_e1=free)   &
  (sig2=green => sig2_e1=free)

OPERATIONS
  reqGreen_e1 =
    IF sig2=red & sig2_e1=free THEN e1:=green END;
  reqGreen_sig0 =
    IF sig3=red & sig0_sig3=free THEN sig0:=green END;
  reqGreen_sig1 =
    IF e0=red & e0_sig1=free THEN sig1:=green END;
  reqGreen_sig2 =
    IF e1=red & sig2_e1=free THEN sig2:=green END;
  reqGreen_sig3 =
    IF sig0=red & sig0_sig3=free THEN sig3:=green END;
```

(End of Exercise 6)
■

5 Conclusion

This chapter presented a compact four-to-six-hour long course on formal verified software development in the B-Method. We do hope that, with additional materials [6] and [7], the chapter provides enough information for an interested reader to go through the course by him or herself. The B-Method was chosen because of the tool support, provided by the freely available Atelier B integrated development environment and prover, and an impressive track record of industrial utilization. This course has been carried out during the Central European Functional Programming (CEFP) summer school in June 2019, in Budapest. A special feature of the course is the utilization of the TD/TS2JC toolset. The centerpiece of the toolset is a modified railway traffic control game, Train Director. It provides a virtual environment for the software developed during the course by its participants, that is for railway controllers. The participants use the toolset at least at the beginning and at the end of the development; at the beginning to examine the scenario for which the controller will be developed and after the development to run the controller with the scenario. A questionnaire given to the participants after the course at the CEFP summer school confirmed the positive impact of the tool set. From 15 participants, 66.7% agreed that the toolset helped them to understand the importance of formal methods and for 99.3% it mattered that they had been able to see their formally developed software running.

The TD/TS2JC toolset had been developed with universality in mind and can be used with any formal method that provides translation to the Java programming language and, also, directly with Java. The chapter tried to demonstrate this universality by describing different types of control modules and corresponding configuration files.

The course presented here includes some pen-and-paper exercises dealing with formal semantics of the B-Method and formal proof. To remind concise, these exercises do not use the formal system behind the B-Method, but try to explain the topic in a way understandable for a common programmer. The course focuses on formulation of the abstract specification and understanding the importance and process of formal verification by mathematical proof. The development to implementation is, deliberately, trivial. A more complex scenario with the specification consisting of multiple components and a nontrivial refinement can be found in [9].

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New Yor (2010)
3. Almeida, J.B., Frade, M.J., Pinto, J.S., De Sousa, S.M.: Rigorous Software Development: An Introduction to Program Verification. Springer, London (2011). https://doi.org/10.1007/978-0-85729-018-2
4. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
5. Fantechi, A.: Twenty-five years of formal methods and railways: what next? In: Counsell, S., Núñez, M. (eds.) SEFM 2013. LNCS, vol. 8368, pp. 167–183. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05032-4_13
6. Korečko, Š.: TD/TS2JavaConn toolset package. https://hron.fei.tuke.sk/korecko/FMInGamesExp/resources/allInOneTDTS2J.zip (2019)
7. Korečko, Š.: Verified software development in B-Method short course package (2019). https://hron.fei.tuke.sk/korecko/cefp19/cefp19BmethodPack.zip
8. Korečko, Š, Sobota, B.: Computer games as virtual environments for safety-critical software validation. J. Inf. Organ. Sci. **41**(2), 197–212 (2017)
9. Korečko, Š., Sorád, J.: Using simulation games in teaching formal methods for software development. In: Queirós, R. (ed.) Innovative Teaching Strategies and New Learning Paradigms in Computer Programming, pp. 106–130. IGI Global (2015)
10. Korečko, Š, Sorád, J., Dudláková, Z., Sobota, B.: A toolset for support of teaching formal software development. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 278–283. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_21
11. Lano, K.: The B Language and Method: A Guide to Practical Formal Development, 1st edn. Springer, New York (1996). https://doi.org/10.1007/978-1-4471-1494-9
12. Larsen, P., Fitzgerald, J., Riddle, S.: Practice-oriented courses in formal methods using vdm++. Formal Aspects Comput. **21**(3), 245–257 (2009). https://doi.org/10.1007/s00165-008-0068-5
13. Liu, S., Takahashi, K., Hayashi, T., Nakayama, T.: Teaching formal methods in the context of software engineering. SIGCSE Bull. **41**(2), 17–23 (2009). https://doi.org/10.1145/1595453.1595457
14. Reed, J.N., Sinclair, J.E.: Motivating study of formal methods in the classroom. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 32–46. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30472-2_3
15. Schneider, S.: The B-Method: An Introduction. Cornerstones of Computing, Palgrave (2001)
16. Spivey, J.M., Abrial, J.: The Z Notation. Prentice Hall Hemel Hempstead, Englewood Cliffs (1992)
17. Train director homepage (2020). https://www.backerstreet.com/traindir/en/trdireng.php
18. Atelier B homepage (2021). https://www.atelierb.eu/en/