Using Simulation Games in Teaching Formal Methods for Software Development

Štefan Korečko, Ján Sorád

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 041 20 Košice, Slovakia, stefan.korecko@tuke.sk, jansorad1@gmail.com

ABSTRACT

Because of the current trend of massification of higher education, motivation of students is a serious issue, especially in courses closely related to mathematics. The ones that undoubtedly belong to this group are courses dealing with formal methods for software development, such as Z notation, B-Method or VDM. The chapter shows how a customized simulation game can be used to bring a domain typical for utilization of formal methods, the railway domain, to students and thus motivate them to learn these sophisticated ways of software development. By means of two examples it demonstrates that such tool, despite of its limited scope, can be used to teach variety of concepts related to formal methods. It also discusses related approaches to teaching formal methods, describes the customized game and its application in teaching and evaluates experiences with the application.

Key words: education, motivation, simulation, game, formal methods, formal verification, formal refinement, B-Method

INTRODUCTION

We live in the era of massification of higher education. We encounter not only highly motivated and interested students but also average ones, where didactic methods, usually used on lower types of schools become relevant. All kinds of subjects in university curricula are affected by this situation but maybe the most suffering ones are those closely related to the field of mathematics. And formal methods courses definitely belong to this group.

Formal methods (FM) 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 defined syntax and semantics and mathematically based means that some mathematical apparatus (formal logic, set theory, etc.) is used to define the language. But as Cerone, Roggenbach, Schlingloff, Schneider and Shaikh (2013) note, a language is not enough to constitute a formal method. To call it a method procedures that allow doing something with specifications written in the language have to be present, too. An example of a well-known FM are regular expressions (Cerone et al., 2013): Syntax of its language can be specified by a context-free grammar. For the semantics there are several ways how to define it, for example by specifying corresponding sets of words or constructing a finite automaton that recognizes words satisfying given expression. A procedure can, for example, be a replacement of every word that satisfies given expression by another word. There are many ways how to classify FM and one, especially interesting from the educational point of view, is a taxonomy based on automation of their procedures and on how easy it is to use them. This taxonomy distinguishes between lightweight and heavyweight formal methods. Lightweight formal methods usually do not require deep expertise. The heavyweight ones are more complex, less automatic, but also more finely grained and powerful (Almeida, Frade, Pinto, & de Sousa, 2011). We can say that for a lightweight FM it is enough to learn its language and know what button to hit in corresponding software tool to do this or that. Often it is not even necessary to learn formal semantics of its language, an explanation in a natural language is sufficient. The aforementioned regular expressions are a lightweight FM. To use them for a text search or replacement in a text editor one just has to read few lines in the editor user's manual, write an expression to an appropriate text field and press a button next to it. On the other hand, significant examples of heavyweight FM are those involving theorem proving as a method of software correctness verification. In principle, the theorem proving cannot be fully automated because underlying theories are usually not decidable. So, to prove assertions about a system a human assistance is often required and to be able to assist one has to possess knowledge about syntax and formal semantics of the language of given FM and operation of its prover. This means a lot of effort but as Harrison (2008) points out, theorem proving brings substantial benefits over other, highly automated, verification methods (e.g. model checking). Provided that properties of a system are correctly specified, its formal verification can ensure that the properties will hold in any state of the system. In an ideal world all software should be like this – 100% verified before its delivery to users. But in reality we use to get faulty software, be it games, operating systems or firmware, and faults are fixed afterwards by means of updates.

As university teachers we sometimes experience resistance from students when a new language or method is introduced, even if it is a widely used one. And position of formal methods courses in software engineering curricula is much worse. Not only are FM too close to the unpopular math but there are not many companies using them in practice. And, especially in the case of the heavyweight ones, we can find them only in specific application areas where their use and cost are justified (Almeida et al., 2011). Of course, we would like to see more widespread utilization of FM and we hope to achieve it by introducing as much students as possible to the art of their application. A big obstacle here is an elective status of many FM courses. So, the essential question is how to motivate students to take FM courses and to stay in them. It is critical to properly choose an application area on which the use of FM will be demonstrated and for which the students will develop something using formal methods. An area where a software fault is able to cause too much damage or loss of lives before any update can be applied. In addition, it should be an area where formal methods have already been successfully applied. According to the comprehensive survey (Woodcock, Larsen, Bicarregui, & Fitzgerald, 2009) and its recent update (Fitzgerald, Bicarregui, Larsen, & Woodcock, 2013) the most of FM industrial success stories can be found in the areas of transportation, finance and defense. What these areas have in common is that they are "physically" out of reach when teaching formal methods. But we have to make them available to students in a believable and funny way. To do this we propose to take existing (simulation) games and modify them in order to allow communication with formally developed software. The games will provide virtual representations of the areas with devices controlled by the software developed using FM. This chapter presents one concrete implementation of the proposal and its use in an undergraduate formal methods course at the home institution of the authors. The implementation offers a virtual railway domain provided by a modified simulation game called Train Director (http://www.backerstreet.com/traindir) and a proxy application, which communicates with Train Director (TD) and allows to load a control module that controls devices (signals and switches) in a scenario simulated in TD. The control module is a Java application, which should, but not have to, be developed by formal methods.

The rest of the chapter is organized as follows. The next section deals with existing approaches to teaching formal methods. The third one is dedicated to the implementation of the proposal itself. It describes important choices the authors made and their reasons, tools that have been modified and developed and use of the implementation in education, including two examples. The fourth section presents other developments of the proposal and ideas for future work. The fifth one deals with the work related to this and the final section concludes with an evaluation of gained experience and cost of developed solutions.

BACKGROUND

The importance of teaching formal methods properly is evident from a number of specialized workshops and meetings, held in association with significant FM conferences and symposia, exclusively or partially dedicated to the FM education. Examples of these are "Teaching Formal Methods" meetings from 2004, 2006 and 2009 or "Fun with Formal Methods" workshop from 2013. And problems related to recruitment to and retention on FM courses is one of the main topics of these events. The reasons of these problems are similar to our situation: To make computer science and software engineering study more accessible FM courses are becoming elective (Reed, & Sinclair, 2004) and it is hard to motivate present-day, practically oriented, students to deal with FM (Larsen, Fitzgerald, & Riddle, 2009), especially considering that FM are used in industry in the most developed countries only (Cristiá, 2006).

A proper choice of examples and their relation to practice is regarded as important by Reed and Sinclair (2004), Larsen, et al. (2009), Liu, Takahashi, Hayashi and Nakayama (2009), Cerone et al. (2013) and many others. Reed and Sinclair (2004) and Liu et al. (2009) advocate for such choice of examples that will clearly show benefits of formal methods, i.e. show what can be achieved by using FM but not by other approaches. Larsen, et al. (2009) agree and present two undergraduate introductory courses, taught at universities in Denmark and UK, where important concepts are illustrated by examples derived from industrial case studies. In the courses presented they focus on lightweight formal methods, or a lightweight use of formal methods with greater possibilities, but they also mention other courses, which deal with a "heavier" stuff like formal verification. Teaching heavyweight FM is the topic of Feinerer and Gernot (2009), who review four tools with respect to their suitability for teaching formal software verification by theorem proving. They state that despite the tools for formal software verification didn't reach the automation level of model checkers used in hardware verification, they have become automated enough to be used more often in the industry. This is supported by the evaluation results in their paper.

In (Cristiá, 2006) the situation in teaching FM in Argentina is described. Its author especially deals with the reasons why to teach formal methods in a country without any industry that uses them and, together with Feinerer and Gernot (2009) shares our belief that FM should be used more often in the industry and that this can be achieved via properly educated students. While virtually all educators stress out an importance of good tool support, Liu et al. (2009) suggest handwriting formal specifications as the best way to learn syntax and semantics of given formal language.

Regarding our proposal, it is important to mention the work of Balz and Goedicke (2010), who implemented an idea similar to it in several aspects. They had took a game-oriented visual simulation environment called Greenfoot (http://www.greenfoot.org/) and added a small framework to it, which allows to embed formally specified software into an application developed and simulated in Greenfoot. The framework supports one formal method, state machines, and these machines are written in annotated Java, similarly to the rest of applications for Greenfoot. In addition, Balz and Goedicke (2010) provide a tool that is able to transform the embedded machines to more abstract models that can be analyzed and verified in the UPPAAL tool (http://www.uppaal.org/).

It should be also noted that introducing games and gaming concepts into higher education is not uncommon nowadays. The Greenfoot tool, mentioned above is one example of this. Its primary role is to teach Java programming to high school students and undergraduates by letting them to develop simple 2D graphical programs like simulations or games. There is even a journal dedicated to the topic of educating in funny ways – the "Transactions on Edutainment", issued as a part of the Springer LNCS series.

RAILWAY SIMULATION IN FORMAL METHODS COURSE

In our effort to increase attractiveness of a formal methods course and to clearly demonstrate importance of FM to students we modified the Train Director game to be a virtual application area for control software developed by FM. The section explains why we implemented our proposal in this way (1st subsection), how the modified game and related proxy application operate (2nd subsection) and how they can be used in teaching process (3rd subsection). The 4th subsection presents concrete examples of control software, developed in a formal method called B-Method. The examples demonstrate that the limited scope of the game is not an obstacle in teaching various aspects of formal methods. The final, fifth, subsection discusses usability of our implementation for other formal methods and other approaches to software engineering.

In this section we use the *italics font shape* when referring to tools that create the virtual application area and the Arial Narrow font for names, methods (operations), variables and other parts of control modules and code in the language of B-Method.

Reasons and Choices

The first choice we made is that of the railway domain. As it was mentioned earlier, the domain, which virtual representation we would like to create should have a history of successful formal methods application. To have a desired motivational effect it also has to be a domain almost every student has experience with and where automated systems in control of human lives or valuable assets already exist. And students should be able to easily imagine being jeopardized by failures of these systems. According to recent surveys (Woodcock et al., 2009; Fitzgerald et al., 2013) FM have been most successfully used in the areas of finance, defense and transportation with transportation being the largest one.

In finance, FM have been applied to various areas, such as transaction processing and electronic cash systems. But these situations are usually not life-threatening and our real-life experience teach us that when something accidentally goes wrong with our finances, we report it to our bank, which usually solves the case in our favor.

The military is without any doubt a good domain for FM application. There are also some really scary stories of computer systems malfunction related to it. For example, on June 3, 1980 U.S. early warning systems had detected multiple incoming Soviet nuclear ballistic missiles and preparations for retaliation started. Fortunately, it was classified as a false alarm. Subsequent investigation identified a faulty computed chip as the cause of the incident. In favor of this domain is also the fact that utilization of formal methods is mandatory for certain classes of military software (e.g. UK Defense Standard 00-55 issue 2). But we are no more living in the cold war, the military service is not mandatory in most countries and young people usually see military systems as something distant, encountered only in computer games and during air shows or military parades.

Transportation domain is a huge one, containing various means of individual and public transport and we can even include space exploration here. The space exploration is an ideal area for FM: software in space probes should be correct as it is difficult if not impossible to fix it. And formal methods have already been used here, for example the SPIN model checker in the development of the Cassini probe or Mars exploration rovers (Woodcock et al., 2009). In addition, it is the “home” of one of the most legendary stories of software failure, often used by formal methods propagators – the crash of the Ariane 5 rocket in 1996, caused by a program that unsuccessfully tried to convert a 64-bit floating point number to a 16-bit signed integer. But most of the Earth inhabitants are not worried by failures of space vehicles – maybe with the exception of cases when some of them fall on their heads.

In air transport there are standards for airborne software that involve formal methods, such as DO-178C/ED-12C. And FM are really used here, too. For example, Airbus used the SCADE Suite from Esterel Technologies (http://www.esterel-technologies.com/) for the development of most of the A380 and A400M critical on-board software (Woodcock et al., 2009). There are many automated systems in airplanes, the term “autopilot” is known for decades. But trained personnel are always present on board, so the perception of threat from computer systems is not that significant. The same is true for flight dispatching. And, in fact, not that much people travel by airplanes.

On the contrary, almost everyone is involved in the road transport. Computer systems are routinely used in cars and signaling on crossroads is automated. Nevertheless, there is always a person controlling a vehicle. Autonomous vehicles already exist but are not used by public. Malfunctioning signaling definitely presents a threat but signals are usually used in urban areas with limited speed, so if there are consequences they are usually not lethal and affect relatively small number of people. In addition, car accidents are quite common and cars are designed with this in mind.

Finally, we got to the domain of our choice, to the railway, which we consider the most suitable one and there are reasons to it. The first is its status of a widely used mean of public transportation almost everyone has an experience with. This is also true for the road transport, but what is specific is that fully automated, driverless, trains already exist and are used by public. It is no wonder: trains are bounded to rails, therefore their behavior is easier to define and control than in the case of road vehicles. Driverless trains operate daily on Line 14 of the Paris metro, CDGVAL Charles de Gaulle airport shuttle rail service or Line 9 of the Barcelona metro. And safety-critical parts of control software for these trains and related equipment, like signaling or platform screen doors, have been developed by B-Method (Abrial, 1996), a heavyweight formal method for verified software development. An evidence of this can be found in (Lecomte et al., 2007), (Lecomte, 2009), (Abrial, 2007), (Boulanger, 2012) or in the surveys mentioned above. Size and mass of trains is in general much bigger than of the road vehicles, therefore they are less controllable. This, together with limitation of their movement because of rails, results in much more significant possible consequences of accidents than in the case of road vehicles. From teacher's point of view an attractive feature of the domain is an ease in which various naturally looking situations can be created: every track layout is a special situation of its own and we can consider different purposes of individual trains, stations or tracks, priority of the trains and so on. It is just necessary to find a game that allows us to build and simulate such layouts.

Fortunately, there are several candidates. In (Korečko, Sorád, & Sobota, 2011) we published results of the search for such game. Two hot candidates were considered – Open Rails (http://www.openrails.org/) and Train Director. Both fulfilled our basic criteria: they allow to build and simulate railway scenarios with working signals and switches, in both it is possible to have more than one train in the simulation and they are open source. There are also big differences between the two. The Train Director focuses on the operation of centralized traffic control. Therefore it provides only schematic 2D representation of the track layout and operation of trains is simplified. On the other hand, Open Rails is a successor of the Microsoft Train Simulator, so it provides nice 3D graphics and sophisticated train models. In 2011 we opted for Train Director because of its relative simplicity and temporal unavailability of the Open Rails source code. Later we also started with modification of the Open Rails, but this is still under development. A question related to this choice is why we didn't implement a new game but modified an existing one. First, we believe that modifying an existing game requires less effort. Second, this way we can use scenarios already developed by others. And there is also a chance that community around the game will help in propagation of formal methods.

At first glance it seems that there is a controversy in our choices: We are trying to improve formal methods education but the tools we offer work with Java applications. And Java is a general purpose programming language, not a formal method or language. But Java is also a language which is supported by code generators of many formal methods tools for software development. It is also a very popular programming language and most software engineering students encounter it during their study. Paradoxically, the formal method we use, the B-Method, doesn't support code generation to Java but we developed our own compiler called BKPI compiler (Korečko, & Dancák, 2011), which is based on the French compiler jBTools (Voisinet, Tatibouet, & Hammad, 2002).

Some may argue that it will be better for our tools to work directly with formal specifications and not only with applications (in Java) generated from them. But we disagree, because the purpose of the tools is to create a virtual application area for formally developed software and not to make its development too simple for students.

Train Director and TS2JavaConn

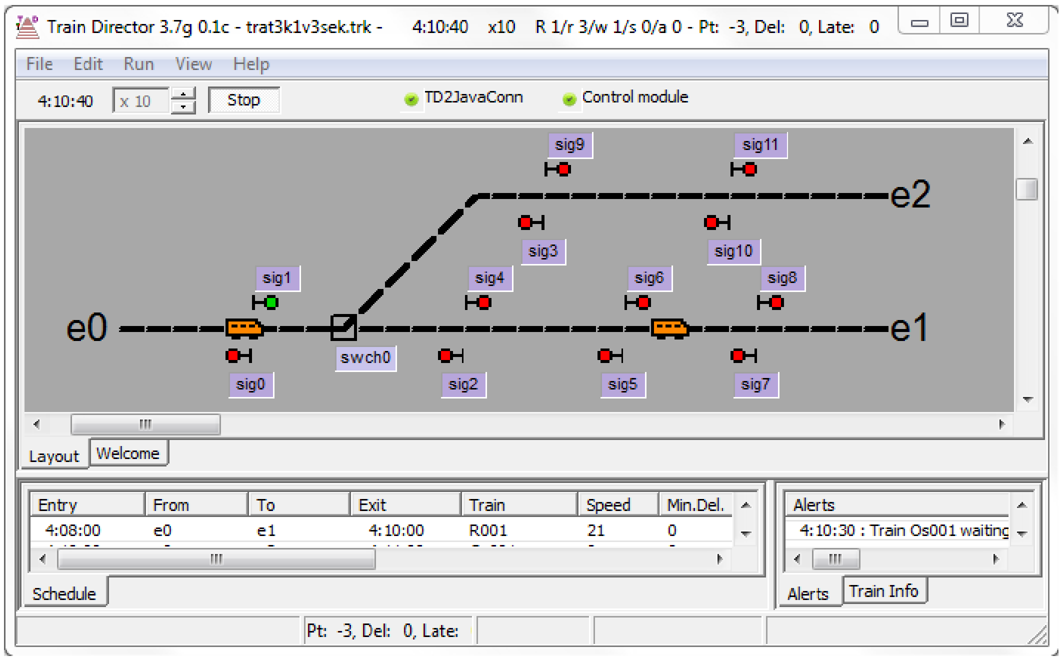
The virtual railway environment is provided by two tools: a modified version of the game Train Director and a newly developed TS2JavaConn, implemented in Java. TS2JavaConn is the proxy application that provides communication between the game and control modules.

Train Director

The *Train Director* (*TD*) is a centralized traffic control simulator. In TD a player can create a railway scenario and simulate it. The scenario consists of a schematic track layout (Fig. 1) and a train schedule, which has a form of a text file. The player's goal during the simulation is to throw switches and clear signals in such a way that the trains will follow the schedule. The game changes some of the signals automatically and prevents collisions. One of the features that persuaded us to choose it is a presence of a simple server interface for external control. While changes in its user interface are almost invisible, internally a lot of code has been modified in order to adjust the game for our purposes. We disabled the internal logic and implemented train collisions. We also added a possibility to name signals and switches and to show the names in the user interface. This was necessary, because without the names it will be impossible to refer to individual signals and switches from the control modules. The build-in server interface was changed, too. The original version worked with messages containing coordinates of mouse clicks, the new one sends and receives detailed information about events and commands and their parameters.

The modified Train Director communicates with TS2JavaConn by means of messages sent via a TCP connection. TD is able to send *request messages* and *informational messages*. A *request message* is sent to TS2JavaConn every time a train stops before a red signal (*requestGreen* message), wants to enter a track layout (*requestDepartureEntry*) or departure from a station (*requestDepartureStation*). These messages also contain data necessary for a corresponding control module. The data include a name of corresponding signal, entry point or station, name of the train and names of following stations the train should visit according to the schedule. An entry point is a named end of a track, where trains can enter or leave the layout. When and where a train enters and should leave the layout is specified in the schedule of the scenario. A station is just a named track section. Whether and for how long a train stops at the station is defined in the schedule, too. *Informational messages* include *sectionLeave*, which is sent to TS2JavaConn when a train leaves current track section and *sectionEnter*, sent when it enters a new one. In the real railway a track can be arbitrarily divided into sections. In Train Director we simplified it in such a way that a track section always starts and ends at some signal, switch or entry point. Train Director also receives messages. These messages are commands from TS2JavaConn to, for example, start or stop a simulation or to change states of signals or switches.

Fig. 1 shows the modified TD during a simulation of a scenario with a simple layout. The layout has three entry points (*e0*, *e1*, *e2*), one switch (*swch0*) and twelve signals (*sig0* to *sig11*). The track sections are named by elements on their ends, from left to right. If there are two signals at the same place (e.g. *sig0* and *sig1* in Fig. 1) then the one guarding given section is used for naming. So, for Fig. 1 we have *e0_sig0*, *sig1_swch0*, *swch0_sig2*, *swch0_sig3* and so on.



Train Director 3.7g 0.1c - trat3k1v3sek.trk - 4:10:40 x10 R 1/r 3/w 1/s 0/a 0 - Pt: -3, Del: 0, Late: 0

Entry	From	To	Exit	Train	Speed	Min.Del.
4:08:00	e0	e1	4:10:00	R001	21	0

Alerts
4:10:30 : Train Os001 waiting

Pt: -3, Del: 0, Late:

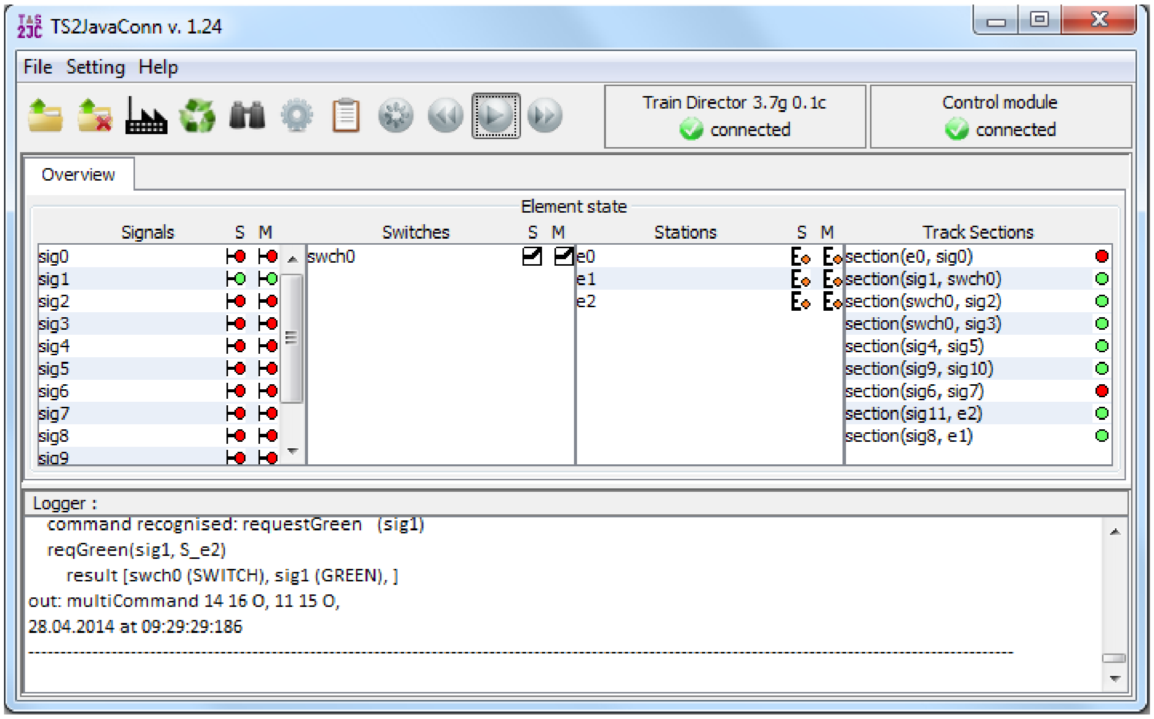
Figure 1. Modified Train Director during simulation.

TS2JavaConn

The development of the second tool, *TS2JavaConn* (Fig. 2), was necessary because Train Director is a C++ application and control modules are in Java. The second reason was that we intend to use it with other railway simulators as well.

The *control modules* are Java applications. Every module has to contain one *"main" class*, which provides interface for communication with the railway scenario it controls (via TS2JavaConn). This interface consists of methods called when the messages are received from Train Director and enquiry methods used to retrieve information about states of track devices from the control module. The enquiry methods are called before the commands to change states of signals or switches are sent to Train Director. Another mandatory part of every module is a *configuration file*, which defines how the methods of the module are mapped to messages from and to Train Director, what are their names, parameters and return values. A wide range of options is available: for input and output parameters we can use primitive types, like integer or Boolean, enumerated sets or mappings. The methods can be non-parametric, where corresponding message parameters are parts of their names or parametric, where they are usual parameters. For example, in a control module for the scenario depicted in Fig. 1 we need twelve methods, i.e. reqGreen_sig0 to reqGreen_sig11, to respond to the *requestGreen* messages for signals in a non-parametric version, but only one method reqGreen(sig) in a parametric one. Examples of both versions are given in the "Examples for B-Method Course" section. It is also possible to read values of control module variables directly and not via the enquiry methods. Number of additional classes and libraries in control modules is not limited, so the modules can be really sophisticated and complex applications A question may be asked why we bother with the non-parametric representation, but our practical experience shows that more complicated data representation, necessary for the parametric one, can make fully automated verification (proofs) impossible even for the simplest scenarios.

TS2JavaConn also provides a GUI where a user can load a control module (first button in the toolbar in Fig. 2) unload the module (2nd button) open a tab with a control module generator (3rd button), reset the connection with TD (4th button) or control the simulation in TD (round buttons). The "Element state" part of the "Overview" tab lists all track elements in the simulated scenario and indicates their state in TD (the column with the title “S” in Fig. 2) and in the control module (the column with the title “M”). The “Logger” part records communication between TD and the control module. The module generator can create the configuration file and control module specification (without bodies of methods or operations) in Java and in languages of formal methods B-Method and Perfect Developer. For Java and B-Method both parametric and non-parametric versions are supported.



Signals	S	M	Switches	S	M	Stations	S	M	Track Sections	
sig0	🟥	🟥	swch0	✓	✓	e0	E	E	section(e0, sig0)	🟥
sig1	🟩	🟩				e1	E	E	section(sig1, swch0)	🟩
sig2	🟥	🟥				e2	E	E	section(swch0, sig2)	🟩
sig3	🟥	🟥							section(swch0, sig3)	🟩
sig4	🟥	🟥							section(sig4, sig5)	🟩
sig5	🟥	🟥							section(sig9, sig10)	🟩
sig6	🟥	🟥							section(sig6, sig7)	🟥
sig7	🟥	🟥							section(sig11, e2)	🟩
sig8	🟥	🟥							section(sig8, e1)	🟩
sig9	🟥	🟥								

```
  command recognised: requestGreen  (sig1)
  reqGreen(sig1, S_e2)
    result [swch0 (SWITCH), sig1 (GREEN), ]
out: multiCommand 14 16 O, 11 15 O,
28.04.2014 at 09:29:29:186
------------------------------------------------------------------------------------------------------------
```

Figure 2. TS2JavaConn during simulation.

Communication Process

In addition to showing the user interface of the tools, Fig. 1 and Fig. 2 also capture an actual communication between Train Director and TS2JavaConn with a control module loaded. The tools are shown exactly after the moment when a request for clearing the signal *sig1* is received from the train *R001*. As we can see in the “Logger” part, TS2JavaConn responds by calling parametric method reqGreen with parameters sig1 and S_e2 from the connected control module. The method changes the values of variables that represent *sig1* and *swch0*. Position of the switch is changed because the destination of *R001* is the entry point *e2*. This is indicated by the value of the second parameter of reqGreen, the S_e2. After the execution of reqGreen is finished, TS2JavaConn reads values of module variables by calling corresponding enquiry methods and sends modified ones to the simulator, using so-called *multiCommand* message. The simulator reacts by setting *sig1* to green and changing the position of *swch0*.

One may wonder why we use the value S_e2 and not e2 for the entry point *e2*. This is because entry points have two meanings; they can be treated as signals or as stations. They are seen as signals when a train enters the layout: the train can enter only after the corresponding entry point is set to green. And they are regarded as stations when they define destinations of trains. The prefix “S_” indicates the station role. The concrete form of the prefix can be set in the configuration file.

Simplifications

Of course, the virtual railway, as represented by the tools, is considerably simplified when compared to the real one. The most significant simplifications are:

1. *Unrealistic train operation.* In Train Director the parameters as length or weight of trains are not considered, so a train fits any section and can stop immediately.
2. *Absolute reliability of trains and track devices.* Trains never disobey signals and signals and switches always operate according to orders given.
3. *Absolute reliability of communication links.* Provided that both tools are initialized properly, all messages are delivered correctly, without any (simulated) loss or corruption of data.
4. *Sequential processing of requests.* The communication between Train Director and TS2JavaConn described above is always executed as one atomic operation. So, if two requests, say *A* and *B*, occur immediately one after the other then *B* is processed only after the results of *A* (i.e. changes in the corresponding control module) manifest in simulated scenario.

It is possible to remove all the simplifications. For example, the reliability can be lowered by introducing randomness to the operation of trains, switches and signals and to the communication between the game and control modules. But for now, we do not intend to do this. At least not for the Train Director. Because even as it is, it provides enough challenges for reliable software development within the scope of an undergraduate FM course.

Utilization in Teaching

The tools, the modified Train Director and TS2JavaConn, can be used during both lectures and practices. A good place to introduce Train Director is a lecture about typical application areas for (heavyweight) formal methods. In this way we can easily show why software correctness is critical there and motivate students to deal with the game before introducing a concrete formal method. The second benefit is important, because, as our experience shows, if the tools and the method are introduced too close to each other, some students tend to focus more on the tools. On lectures dedicated to a concrete formal method the tools can be used for examples demonstrating various aspects of the method (two such examples for B-Method are shown in the next subsection). The advantage here is time saved because it is not needed to explain context of each example. On the other hand we do not recommend showing only examples prepared with the tools. Otherwise students can get an impression that formal methods are all about railway.

While the use of the tools on lectures is beneficial, the best place for them is on practices, where students are given assignments to develop a dependable control module for a railway scenario. The corresponding teaching process usually looks like follows:

1. *Scenario creation.* A teacher creates a new scenario or modifies an existing one in Train Director. He should be aware that not all types of track devices can be used, only simple red/green signals and two-way switches.
2. *Scenario analysis.* The scenario is presented to a student with a task to develop a controller for it. The student analyzes the scenario by “playing” with it in Train Director. Both original and modified versions of Train Director can be used for this task. If the modified Train Director is disconnected from TS2JavaConn, switches and signals can be operated manually.
3. *Control module development.* After getting familiar with the scenario the student develops the control module itself, using given formal method and tools available for it. The student should start with data representation of elements from the scenario (i.e. track sections, switches, signals, stations, trains) and formalization of safety requirements, continue with writing operations (methods), verification and refinement tasks and finish with code generation. Provided that his

FM is supported, the student can use the module generator of TS2JavaConn to create the configuration file and an empty main component of the module. The teacher assists during the development, if necessary.

4. *Simulation.* The student loads compiled module into TS2JavaConn and simulates corresponding scenario with the module in control. The teacher can provide alternative schedules for the scenario to find out whether the module is really correct.

As it can be seen from the process, our tools really act as a virtual domain and nothing more (except for the module generator). They are intensively used in phases 1, 2 and 4 but during the development itself the student relies on the standard tools available for given formal method. This fulfills our intention to emulate real development process as close as possible.

Examples for B-Method Course

This subsection presents two examples, two control modules, specified in the language of B-Method, which highlight different features of this formal method. The first module is intended for an introductory lecture, is very simple and concentrates on basic capabilities of B-Method and formalization of safety requirements. The second one is composed from several components, including parametric ones, and uses different data representation as the first one. One of its purposes is to show reusability in B-Method. The first module uses nonparametric operations (methods), the second uses parametric ones.

B-Method

B-Method (B) (Abrial, 1996) is a state based model-oriented heavyweight formal method for software development. Its strength lies in a well-defined development process, which allows specifying a software system as a collection of components, called B-machines, and refining such an abstract specification to a concrete one. The concrete specification can be automatically translated to a general purpose 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). All components in B are written in its own B-language, a combination of the Zermelo-Fraenkel set theory and the Guarded Command Language (Dijkstra, 1976). There is an industrial-strength software tool Atelier B (http://www.atelierb.eu/), which supports the whole development process, including proving and code generation to C and ADA. Proofs in Atelier B can be done in fully automatic or human-assisted (interactive) mode. Another useful tool for B-Method is ProB (Leuschel, & Butler, 2003). It allows to animate and model check specifications written in B-language. By animation we mean running or simulation of formal specification, despite the fact that it contains unimplementable features like nondeterminism and non-termination. The B-machine is a component consisting of several clauses:

```
MACHINE M(p)
CONSTRAINTS C
SETS St
CONSTANTS k
PROPERTIES Bh
VARIABLES v
DEFINITIONS D
INVARIANT I
ASSERTIONS A
INITIALISATION T
OPERATIONS
 y<--op(x) =   PRE P THEN S END
```

The most important are MACHINE clause with a name M of the machine and a list p of its formal parameters, the VARIABLES (or CONCRETE_VARIABLES) containing a list v of state variables, INVARIANT with properties I of the state variables, INITIALISATION with an operation T that establishes an initial state of the machine and OPERATIONS that contains its operations. We say that the machine is internally consistent if I holds in each of its states. St is a list of deferred and enumerated sets. They represent new types. Constants of the machine are listed in k and a predicate Bh defines properties of St and k. D is a list of macro definitions and A is a list of lemmas used to simplify proof of the PObs. Only the MACHINE clause is mandatory.

Every operation has a header and a body. The header includes its name (op) and optional input and output parameters (x, y). The body is written in the Generalized Substitution Language (GSL), a part of the B-language. GSL contains several constructs, or "commands", called generalized substitutions (GS). They include:

- x := e. Assignment of a value of expression e to variable x.
- S1 ; S2. Sequential composition: do GS S1 then GS S2.
- S1 || S2. Parallel composition: do S1 and S2 at once.
- PRE E THEN S1 END. Preconditioning. It executes S1 if E holds. Otherwise it doesn't terminate.
- IF E THEN S1 ELSE S2 END. Conditional statement: if E holds, do S1, otherwise do S2. In B this is not a basic GS but a combination of two other GS.

The list is not complete; GS that are not used in examples below are excluded. The formal semantics of GSL is defined by the weakest pre-condition calculus (Dijkstra, 1976). Standardly, the body has the form of PRE GS, however if P is TRUE then it consists only of S ("PRE P" is replaced by "BEGIN").

The PObs for B-machine assert that T always establishes an initial state in which I holds and that for each operation op it holds that if op is executed from a state satisfying I and P then it always terminates in a state satisfying I.

The development process of B is a verified stepwise refinement, where an abstract specification, consisting of B-machines (MM), is modified in one or more steps into an implementable one and correctness of each step is proved. There are two additional components used during the process – Refinement (RR) and Implementation (II). Structures of MM, RR and II are similar, but there are some differences. For example GS ";" and loops are not allowed in MM and "||" and PRE are not allowed in II. A RR or II can refine only one MM or RR but one MM or RR can be refined by more RR or II . To refine means to modify data or operations. Interfaces (i.e. parameters and operation headers) of a refining and a refined component have to be the same. Invariant of RR or II defines not only properties of its variables but also a relation between its variables and variables of the component it refines. Proof obligations of RR and II are similar to those of MM, but take into account operations and variables of the components they refine.

In II we have to use CONCRETE_VARIABLES clause instead of VARIABLES. Concrete means implementable and therefore not all types are allowed. We can also use the CONCRETE_VARIABLES clause in MM and RR. Then variables listed in this clause remain the same in all subsequent refinements and we do not need to list them again.

As it was mentioned above, a specification in B usually consists of more than one component. To access contents of one component from another one several composition mechanisms can be used. For example, SEES and USES allow different level of read-only access, INCLUDES allows to call operations of accessed component in the accessing one and IMPORTS has the same meaning as INCLUDES, but for implementations. These mechanisms are usually defined right after the CONSTRAINTS or REFINES clause.

Example 1: For One Lecture

This introductory example is a control module for a very simple scenario (Fig. 3) with two signals (*sig0*, *sig1*), two entry points (*e0*, *e1*) and two sections (*e0_sig0*, *sig1_e1*). The specification of the module consists of two components – the B-machine trat1k and its refinement, the implementation trat1k_i.

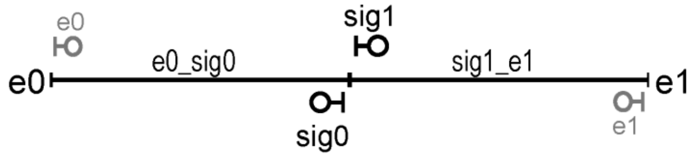
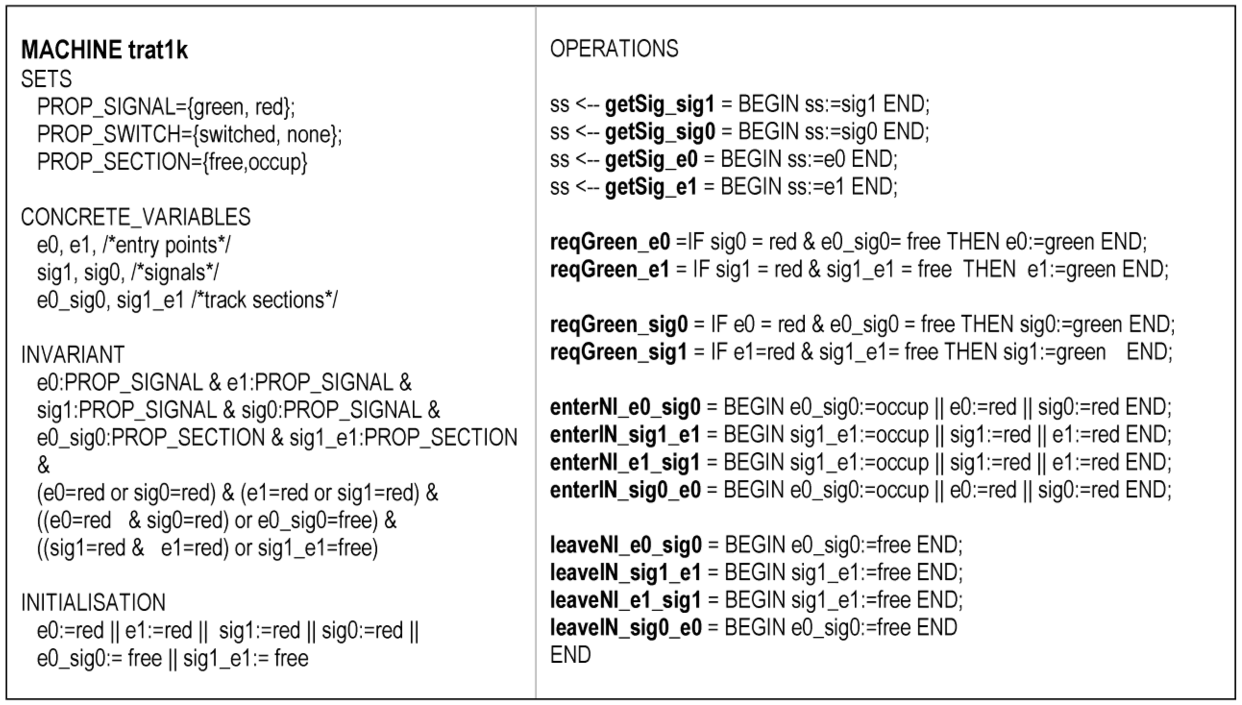


Figure 3. Track layout from example 1 with names of sections and signals.



```
MACHINE trat1k
SETS
  PROP_SIGNAL={green, red};
  PROP_SWITCH={switched, none};
  PROP_SECTION={free,occup}

CONCRETE_VARIABLES
  e0, e1, /*entry points*/
  sig1, sig0, /*signals*/
  e0_sig0, sig1_e1 /*track sections*/

INVARIANT
  e0:PROP_SIGNAL & e1:PROP_SIGNAL &
  sig1:PROP_SIGNAL & sig0:PROP_SIGNAL &
  e0_sig0:PROP_SECTION & sig1_e1:PROP_SECTION
  &
  (e0=red or sig0=red) & (e1=red or sig1=red) &
  ((e0=red   & sig0=red) or e0_sig0=free) &
  ((sig1=red &   e1=red) or sig1_e1=free)

INITIALISATION
  e0:=red || e1:=red ||  sig1:=red || sig0:=red ||
  e0_sig0:= free || sig1_e1:= free
```

```
OPERATIONS

ss <-- getSig_sig1 = BEGIN ss:=sig1 END;
ss <-- getSig_sig0 = BEGIN ss:=sig0 END;
ss <-- getSig_e0 = BEGIN ss:=e0 END;
ss <-- getSig_e1 = BEGIN ss:=e1 END;

reqGreen_e0 =IF sig0 = red & e0_sig0= free THEN e0:=green END;
reqGreen_e1 = IF sig1 = red & sig1_e1 = free  THEN  e1:=green END;

reqGreen_sig0 = IF e0 = red & e0_sig0 = free THEN sig0:=green END;
reqGreen_sig1 = IF e1=red & sig1_e1= free THEN sig1:=green   END;

enterNI_e0_sig0 = BEGIN e0_sig0:=occup || e0:=red || sig0:=red END;
enterIN_sig1_e1 = BEGIN sig1_e1:=occup || sig1:=red || e1:=red END;
enterNI_e1_sig1 = BEGIN sig1_e1:=occup || sig1:=red || e1:=red END;
enterIN_sig0_e0 = BEGIN e0_sig0:=occup || e0:=red || sig0:=red END;

leaveNI_e0_sig0 = BEGIN e0_sig0:=free END;
leaveIN_sig1_e1 = BEGIN sig1_e1:=free END;
leaveNI_e1_sig1 = BEGIN sig1_e1:=free END;
leaveIN_sig0_e0 = BEGIN e0_sig0:=free END
END
```

Figure 4. Source code of B-machine trat1k.

The machine trat1k is shown in Fig. 4. To simplify the matter as much as possible its variables (clause CONCRETE_VARIABLES) match the track devices. Types of the variables are defined in the first three lines of the INVARIANT clause and they are enumerated sets, listed in the SETS clause. The operator "&" is conjunction, "or" is disjunction and ":" means "belongs to". The last three lines of the INVARIANT are safety conditions, formalized from safety requirements on the module. The INITIALISATION sets all signals to red and all sections to free. Entry points are treated as signals only. There are 16 nonparametric operations. The first four are enquiry operations and just return values of corresponding variables. It is also possible to have getSec operations, returning occupation status of sections, but they are not mandatory as the occupation of sections is not used on the Train Director side. The next two are to respond to the *reqestDepartureEntry* messages from Train Director and the two after them to the *requestGreen* messages. The next four respond to the *sectionEnter* messages and the last four to the *sectionLeave* messages. The capital letter "N" means "eNtry point" and "I" means "sIgnal". We have two sections but there are four operations in each of the last two groups. This is to be able to treat an entry or leave from each side differently (albeit it is not necessary in this case).

The implementation trat1k_i is not shown because it is nearly identical to the machine trat1k. It just doesn't contain clauses CONCRETE_VARIABLES and INVARIANT and "||" is replaced by ";". After translation of trat1k_i to Java a class trat1k is created with methods named exactly as operations in trat1k (and trat1k_i). This class is then loaded in TS2JavaCon as a control module for the scenario from Fig.3.

The example is very suitable for explaining how safety requirements can be formalized into the invariant. Here we have two safety requirements:

1. *Only one of the signals guarding a section can be green.* This is formalized to the condition "e0=red or sig0=red" for *e0_sig0* and to "e1=red or sig1=red" for *sig1_e1*.
2. *A signal guarding a section can be green only if the section is free.* This is formalized to the condition "(e0=red & sig0=red) or e0_sig0=free" for *e0_sig0* and to "(sig1=red & e1=red) or sig1_e1=free" for *sig1_e1*.

Thanks to the trivial data representation it is easy to connect informal requirements to corresponding logical assertions (conditions).

An example like this also provides an ideal opportunity to show what FM like B can do and what they cannot do. Actually, the only extra thing B can do is to verify (prove) that the conditions we wrote in the INVARIANT clause hold in every possible state of given system. This is very valuable but it is not enough to guarantee correctness, because there is no way B-Method can verify that the conditions in the INVARIANT are correct. As Bowen and Hinchey (2006) note: "If the organization has not built the right system (validation), no amount of building the system right (verification) can overcome that error." For example, the aforementioned software of the crashed Ariane 5 rocket was functioning perfectly, in accordance with corresponding requirements. But these requirements were wrong (Le Lann, 1997). In fact, they were wrong because they were taken, together with the software itself, from the previous version of the rocket, Ariane 4. Back to our example, we can easily modify the safety requirements and operations in such a way the module will be fully verified but nevertheless causes an accident. Another thing B cannot guarantee is correctness of the interface. We can show that if we interchange bodies of the operations in trat1k, excluding the first four ones, its invariant will still hold.

Example 2: Introducing Advanced Concepts

The second module controls the scenario already shown in Fig.1. It is possible to define a module for this scenario in the exactly same way as the previous one, but here we would like to introduce more advanced concepts of B-Method. Its primary purpose is to show how to use compositional mechanisms to design a control module with reusable components. It also uses the DEFINITIONS clause and a separate file with definitions to make specifications more readable while using primitive data types. Operations are parametric and data representation no more corresponds exactly to the devices in the scenario. Some variables have mappings as types. Verification of this module requires small amount of interactive proving.

The layout of the scenario is divided into four parts (Fig.5 c). Each part is controlled by a separate machine. There are three CntrlSimpleTrack machines and one Cntrl3WayTrack machine. To distinguish between the three machines they are named – trN, trS and trD. All are included in one "main" machine Controller3way3secTr. States of the devices are represented as values 0 and 1, but meaningful aliases are assigned to them in definitions in a separate file values.def (Fig. 6a). This file is then linked to each component of the module. The operator "==" means "rewrites to". Composition of the module is shown in Fig. 5 d). Ellipses are B-machines, rectangles are implementations.

The machine CntrlSimpleTrack and its implementation CntrlSimpleTrack_i represent a controller for a straight track of arbitrary length (measured in sections). Layout of the track with data representation of its elements is shown in Fig. 5 b) and specification of the machine in Fig. 6 b). The operator "-->" means total function, "=>" is implication, "*" is Cartesian product, "{x}" is a set consisting of an element x, "!" is "for all", "x..y" is an integer interval from x to y and "f(x)" is a value of a function f in x. All three variables are in fact arrays of the length length, storing values 0 and 1. All operations are parametric. The first three are enquiry operations; next two react to requests for green signals the sixth to entering a section and the last one to leaving a section. The capital letter "R" means that the corresponding operation deals with signals for trains heading from left to Right, "L" for trains in opposite direction.

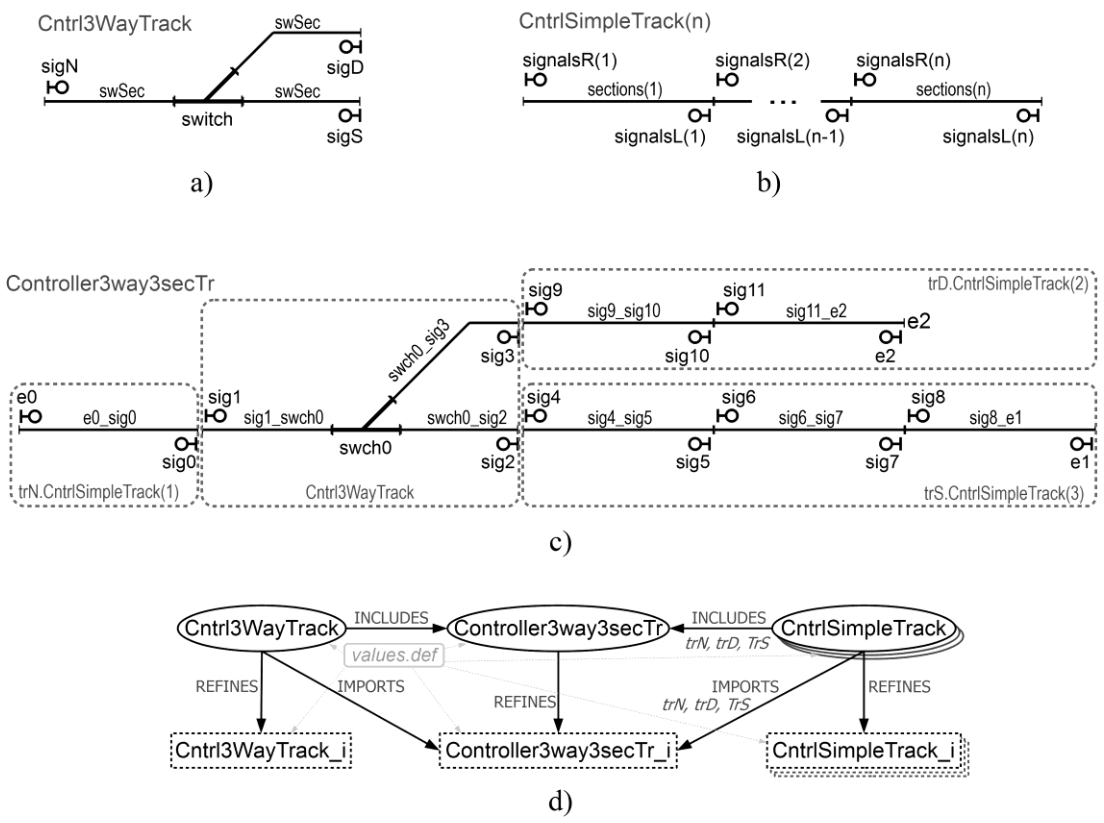


Figure 5. Track layout from example 2 with names of sections, switches and signals (a-c): whole layout (c), component with switch (a) and straight track component (b). Composition of B specification components of corresponding control module (d).

The control module for the part with switch (Fig. 5 a) is specified in the machine Cntrl3WayTrack (Fig. 7) and its implementation Cntrl3WayTrack_i. The style of these specifications is similar to the previous example, with three exceptions. First, all three sections are represented as one "logical" section by the variable swSec. Second, the operation reqGreenN has a parameter dir to determine how the position of the switch should be changed. Third, there is only one operation for section entering (enter3w) and one for leaving (leave3wAll) and leave3wAll should be called only when a train leaves the entire part.

```
DEFINITIONS  red==0; green==1;diverg==0; straight==1;free==0; occup==1
```

(a)

```
MACHINE CntrlSimpleTrack(length)
CONSTRAINTS length:NAT & length>0
DEFINITIONS "values.def"

CONCRETE_VARIABLES sections, signalsR, signalsL
INVARIANT
  sections : (1..length)-->(0..1) &
  signalsR : (1..length)-->(0..1) &
  signalsL : (1..length)-->(0..1)
  &
  !xx.((xx:(1..length)) =>
       ((signalsR(xx)=red) or (signalsL(xx)=red)))
  &
  !xx.((xx:(1..length)) =>
      ((signalsR(xx)=red & signalsL(xx)=red)
        or sections(xx)=free))

INITIALISATION
  sections := (1..length)*{free} ||
  signalsR := (1..length)*{red}  ||
  signalsL := (1..length)*{red}
```

```
OPERATIONS
res<--getSignalR(no) = PRE no:1..length THEN res:=signalsR(no) END;
res<--getSignalL(no) =  PRE no:1..length THEN res:=signalsL(no) END;
res<--getSection(no) =  PRE no:1..length THEN res:=sections(no) END;

reqGreenR(sigNo)= PRE sigNo:1..length THEN
   IF signalsL(sigNo)=red & sections(sigNo)=free THEN
      signalsR(sigNo):=green END
END;

reqGreenL(sigNo)= PRE sigNo:1..length THEN
  IF signalsR(sigNo)=red & sections(sigNo)=free THEN
     signalsL(sigNo):= green  END
END;

enterSec(secNo)= PRE secNo:1..length THEN
   sections(secNo):=occup ||
   signalsR(secNo):=red || signalsL(secNo):=red
END;

leaveSec(secNo)= PRE secNo:1..length THEN sections(secNo):=free END
END
```

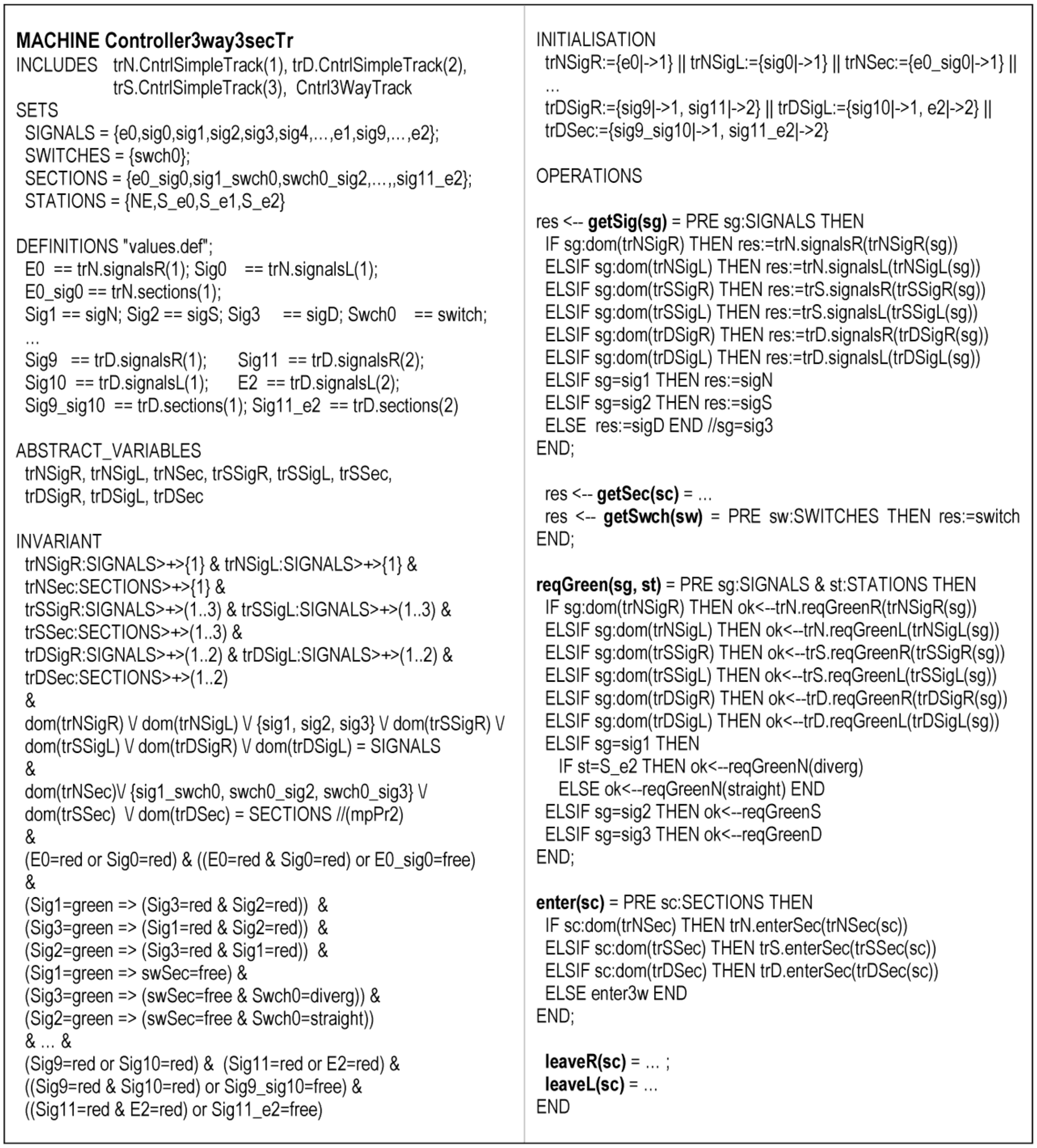
(b)

Figure 6. Source code of definitions file values.def (a) and B-machine CntrlSimpleTrack (b).

```
MACHINE Cntrl3WayTrack
DEFINITIONS "values.def"
CONCRETE_VARIABLES
  sigN, sigS, sigD, swSec, switch

INVARIANT
  sigN:0..1 & sigS:0..1 & sigD:0..1 & swSec:0..1 &
  switch:0..1 &
  (sigN=green => (sigD=red & sigS=red)) &
  (sigD=green => (sigN=red & sigS=red))  &
  (sigS=green => (sigD=red & sigN=red))  &
  (sigN=green => swSec=free) &
  (sigD=green => (swSec=free & switch=diverg)) &
  (sigS=green => (swSec=free & switch=straight))

INITIALISATION
  sigN:=red || sigS:=red || sigD:=red || swSec:=free ||
  switch := straight

OPERATIONS

res<--getSigN = BEGIN res:=sigN END;
res<--getSigS = BEGIN res:=sigS END;
```

```
res<--getSigD = BEGIN res:=sigD END;
res<--getSwitch=BEGIN res:=switch END;
res<--get3wSec =BEGIN res:=swSec END;

reqGreenN(dir)= PRE dir:0..1 THEN
  IF sigS=red & sigD=red & swSec=free
  THEN sigN:=green || switch:=dir END
END;

reqGreenD=
  IF sigN=red & sigS=red & swSec=free
  THEN sigD:=green || switch:=diverg END;

reqGreenS=
  IF sigN=red & sigD=red & swSec=free
  THEN sigS:=green || switch:=straight END;


enter3w= BEGIN
  sigN:=red || sigS:=red || sigD:=red || swSec:=occup END;

leave3wAll = BEGIN swSec:=free END
END
```

Figure 7. Source code of B-machine Cntrl3WayTrack.



```
MACHINE Controller3way3secTr
INCLUDES   trN.CntrlSimpleTrack(1), trD.CntrlSimpleTrack(2),
           trS.CntrlSimpleTrack(3),  Cntrl3WayTrack
SETS
  SIGNALS = {e0,sig0,sig1,sig2,sig3,sig4,…,e1,sig9,…,e2};
  SWITCHES = {swch0};
  SECTIONS = {e0_sig0,sig1_swch0,swch0_sig2,…,,sig11_e2};
  STATIONS = {NE,S_e0,S_e1,S_e2}

DEFINITIONS "values.def";
  E0  == trN.signalsR(1); Sig0    == trN.signalsL(1);
  E0_sig0 == trN.sections(1);
  Sig1 == sigN; Sig2 == sigS; Sig3     == sigD; Swch0    == switch;
  …
  Sig9   == trD.signalsR(1);      Sig11  == trD.signalsR(2);
  Sig10  == trD.signalsL(1);      E2  == trD.signalsL(2);
  Sig9_sig10  == trD.sections(1); Sig11_e2  == trD.sections(2)

ABSTRACT_VARIABLES
  trNSigR, trNSigL, trNSec, trSSigR, trSSigL, trSSec,
  trDSigR, trDSigL, trDSec

INVARIANT
  trNSigR:SIGNALS>+>{1} & trNSigL:SIGNALS>+>{1} &
  trNSec:SECTIONS>+>{1} &
  trSSigR:SIGNALS>+>(1..3) & trSSigL:SIGNALS>+>(1..3) &
  trSSec:SECTIONS>+>(1..3) &
  trDSigR:SIGNALS>+>(1..2) & trDSigL:SIGNALS>+>(1..2) &
  trDSec:SECTIONS>+>(1..2)
  &
  dom(trNSigR) \/ dom(trNSigL) \/ {sig1, sig2, sig3} \/ dom(trSSigR) \/
  dom(trSSigL) \/ dom(trDSigR) \/ dom(trDSigL) = SIGNALS
  &
  dom(trNSec)\/ {sig1_swch0, swch0_sig2, swch0_sig3} \/
  dom(trSSec)  \/ dom(trDSec) = SECTIONS //(mpPr2)
  &
  (E0=red or Sig0=red) & ((E0=red & Sig0=red) or E0_sig0=free)
  &
  (Sig1=green => (Sig3=red & Sig2=red))  &
  (Sig3=green => (Sig1=red & Sig2=red))  &
  (Sig2=green => (Sig3=red & Sig1=red))  &
  (Sig1=green => swSec=free) &
  (Sig3=green => (swSec=free & Swch0=diverg)) &
  (Sig2=green => (swSec=free & Swch0=straight))
  & … &
  (Sig9=red or Sig10=red) &  (Sig11=red or E2=red) &
  ((Sig9=red & Sig10=red) or Sig9_sig10=free) &
  ((Sig11=red & E2=red) or Sig11_e2=free)
```

```
INITIALISATION
  trNSigR:={e0|->1} || trNSigL:={sig0|->1} || trNSec:={e0_sig0|->1} ||
  …
  trDSigR:={sig9|->1, sig11|->2} || trDSigL:={sig10|->1, e2|->2} ||
  trDSec:={sig9_sig10|->1, sig11_e2|->2}

OPERATIONS

res <-- getSig(sg) = PRE sg:SIGNALS THEN
  IF sg:dom(trNSigR) THEN res:=trN.signalsR(trNSigR(sg))
  ELSIF sg:dom(trNSigL) THEN res:=trN.signalsL(trNSigL(sg))
  ELSIF sg:dom(trSSigR) THEN res:=trS.signalsR(trSSigR(sg))
  ELSIF sg:dom(trSSigL) THEN res:=trS.signalsL(trSSigL(sg))
  ELSIF sg:dom(trDSigR) THEN res:=trD.signalsR(trDSigR(sg))
  ELSIF sg:dom(trDSigL) THEN res:=trD.signalsL(trDSigL(sg))
  ELSIF sg=sig1 THEN res:=sigN
  ELSIF sg=sig2 THEN res:=sigS
  ELSE  res:=sigD END //sg=sig3
END;

  res <-- getSec(sc) = …
  res  <--  getSwch(sw)  =  PRE  sw:SWITCHES  THEN  res:=switch
END;

reqGreen(sg, st) = PRE sg:SIGNALS & st:STATIONS THEN
  IF sg:dom(trNSigR) THEN ok<--trN.reqGreenR(trNSigR(sg))
  ELSIF sg:dom(trNSigL) THEN ok<--trN.reqGreenL(trNSigL(sg))
  ELSIF sg:dom(trSSigR) THEN ok<--trS.reqGreenR(trSSigR(sg))
  ELSIF sg:dom(trSSigL) THEN ok<--trS.reqGreenL(trSSigL(sg))
  ELSIF sg:dom(trDSigR) THEN ok<--trD.reqGreenR(trDSigR(sg))
  ELSIF sg:dom(trDSigL) THEN ok<--trD.reqGreenL(trDSigL(sg))
  ELSIF sg=sig1 THEN
    IF st=S_e2 THEN ok<--reqGreenN(diverg)
    ELSE ok<--reqGreenN(straight) END
  ELSIF sg=sig2 THEN ok<--reqGreenS
  ELSIF sg=sig3 THEN ok<--reqGreenD
END;

enter(sc) = PRE sc:SECTIONS THEN
  IF sc:dom(trNSec) THEN trN.enterSec(trNSec(sc))
  ELSIF sc:dom(trSSec) THEN trS.enterSec(trSSec(sc))
  ELSIF sc:dom(trDSec) THEN trD.enterSec(trDSec(sc))
  ELSE enter3w END
END;

  leaveR(sc) = … ;
  leaveL(sc) = …
END
```

Figure 8. Source code of B-machine Controller3way3secTr.

All is put together in the machine Controller3way3secTr (Fig. 8) and its implementation Controller3way3secTr_i. This machine includes the previous machines for individual parts and defines sets to represent track devices in the entire layout. The entry points are present in both SIGNALS and STATIONS, in the latter with the prefix "S_". The value "NE" represents an undefined station. Additional definitions are added to assign alternative, more meaningful names, to variables from included machines. The variables of Controller3way3secTr map names of track devices to indices of arrays in included CntrlSimpleTrack machines. The first three operations are enquiry operations returning status of signals, sections and switches. The operation responding to *reqestDepartureEntry* and *requestGreen* messages is reqGreen with two parameters, sg and st. The first one is a name of a signal or an entry point, the second is the destination of given train. The second parameter is used to set the switch. The operation enter responds to entering a section, no matter from which direction. For the *sectionLeave* messages we need two operations, one for each direction. This is because of the specific nature of leave3wAll from Cntrl3WayTrack.

To make proving easier a PROPERTIES clause that repeats definition of sets from SETS can be added to Controller3way3secTr. The content of these two clauses will be the same. Only ";" will be replaced by "&". There are no alternative names for track sections sig1_swch0, swch0_sig2, swch0_sig3 in the DEFINITIONS clause. Instead of them we refer directly to variable swSec from Cntrl3WayTrack.

In addition to the utilization of composition mechanisms and reusable components the example also shows that sometimes an approach unnatural at first sight has to be used to achieve effective performance from tools of given FM (the Atelier B's prover in this case). Namely, one can easily see that abstract variables of the machine Controller3way3secTr are in fact constants. So, the ABSTRACT_CONSTANTS clause will be more appropriate for them. But, if we define them as constants then their properties (8th to 12th line in the INVARIANT clause) will require a use of the interactive prover. To be sure that they remain constant we can repeat the content of the INITIALISATION clause in INVARIANT with "||" replaced by "&". Adjusting specifications to tools is not that uncommon in the world of FM, Feinerer and Gernot (2009) report a similar experience with Perfect Developer.

This and the previous example primarily aim at verification and not that much at refinement and abstraction (which is, for example, in contrary to (Larsen et al., 2009)). However, it is possible to explore refinement and abstraction more, especially with reusable components. For example, the component Cntrl3WayTrack has a potential for data refinement: on the most abstract level (B-Machine) we can represent sections exactly as in corresponding track layout (i.e. 3 sections) and then refine it to one "logical" section and thus reduce memory requirements.

Another common feature of both modules is that they always allow access to only one section. A negative consequence of this is that deadlocks can occur when trains from opposite directions meet on the same straight track. They will not collide but they cannot move anymore. But nothing prevents us to build more sophisticated, deadlock-free modules. This opens new possibilities, such as an introduction of recursive functions and loops to components with array-like variables (e.g. CntrlSimpleTrack).

Beyond B-Method

Because of the maturity of its tools and a reputation of an industrially used FM, the B-Method is our primary choice when developing control modules. But thanks to the configurable interface and the fact that modules connect on the Java application level the TS2JavaConn and modified Train Director can be used with ones developed by other formal methods as well. To demonstrate it we developed control modules with three other formal methods, which offer Java code generator – the Vienna Development Method (VDM), namely its object-oriented version VDM++ (Fitzgerald, Larsen, Mukherjee, Plat, & Verhoef, 2005), the language Perfect (Crocker, 2004) and Event-B (Abrial, 2010). For VDM++ the build-in compiler of VDM++ Toolkit (http://www.vdmtools.jp) was used to generate Java code. Similarly, the code from specification in Perfect was generated from its tool Escher Verification Studio (http://www.eschertech.com/products/). There are two Java code generators for Event-B and its software platform Rodin (http://www.event-b.org) and we used EB2J (http://eb2all.loria.fr). All these modules use primitive types for message parameters and non-parametric operations (methods). The Java code generated from Escher Verification Studio and VDM++ Toolkit worked without modifications; in the EB2J case it was necessary to add an explicit constructor. Creation of empty nonparametric module in the language Perfect is supported by TS2JavaConn.

Of course, there is no need to use formal methods at all when developing the control modules; they can be programmed in Java only. In this way the tools can be used to teach testing or writing programs annotated with Java Modeling Language (www.jmlspecs.org/) specifications. The tools can be also suitable for introducing high school students or pupils into programing, maybe even with a gentle introduction to safety issues.

FUTURE DEVELOPMENT

There are several ways in which we intend to further develop the ideas and results presented in this chapter. The first one is to successfully finish modifications of the Open Rails (OR) train simulator to provide functionality at least similar to the modified Train Director. Being 3D simulator with sophisticated model of train operation, OR offers much more possibilities of what to control and how to control than Train Director. For example, we can allow a control module to take charge of not only track equipment but also the trains. In OR trains do not stop instantly before signals but obey laws of physics, so we can incorporate parameters such as distance from a signal and speed and weight of trains into the modules. OR is compatible with Microsoft Train Simulator (MSTS), therefore routes designed for MSTS can be used in OR and there are many of them, realistically capturing railways from all over the world. This means a lot of material for control module creation. OR also offers a multiplayer mode. It will be interesting to see how successful the control modules are when trains are operated by unpredictable human players.

Another way in which the "railway control" idea can be explored is to go physical. Namely, to use digitally controlled model railway instead of the simulated one. To possess such an installation can be interesting for promotional purposes. Formally developed control modules can be also used in automated systems based on hardware platforms like Arduino or Intel Galileo.

We also intend to utilize the competitive nature of human beings to propagate formal methods. On the basis of our experiences with Train Director we are developing an online platform, connected to social networks, which will be able to host various simulators, representing various domains, and control modules for scenarios simulated in them. Users of the platform will have the option to prepare (challenging) scenarios or control modules for them.

Scenes or games created for existing gaming and visualization engines can be also used as virtual application areas for formally developed software. We would like to continue in our experiments with the Unity engine (https://unity3d.com/), which can be used for free and offer scripting in C#, the language supported by several formal methods tools. We already made a visualization of the Boiler case study from (Abrial, 1996) and modified several existing scenes to be able to host control modules.

Other areas worth exploration include population of agent systems by formally developed agents or entering a gaming competition, such as the StarCraft AI Competition, with a verified bot.

RELATED WORK

The perception of problems with formal methods education here is similar to opinions of other educators and researchers, but the solution offered differs. We follow the same basic idea as Larsen et al. (2009) that FM should be demonstrated on examples derived from industrial practice, but contrary to them we focus on one domain and provide tools that create virtual representation of the domain. While the approach we have used is strongly tool-oriented, it can be used in accordance with Liu et al. (2009), who prefer handwriting formal specifications, at least when learning the language of given method: Specifications of simple control modules, like the one from Example 1, can be written by hand on the basis of experimentation with corresponding scenario in Train Director within one practice. Our approach also follows principles of teaching formal methods, defined by Cerone et al. (2013). Most notably, it allows to focus on one formal method (principle 1) with industrial-strength tool support (principles 3 and 5) and provides a stable platform for lab classes (principle 6). Moreover, we picked up the railway domain, which is familiar to students from everyday life and Java as the target language because students usually know it well from other courses (principle 7). And the approach stimulates not only exploration of syntax and semantics of the formal method taught but also associated procedures of formal verification and refinement (principle 8). Finally, playing with trains is fun (principle 10).

The work most similar to ours is (Balz, & Goedicke, 2010), where the game-oriented visual simulation environment Greenfoot (http://www.greenfoot.org/), intended for teaching object orientation with Java, has been enhanced by a small framework that allows to embed state machines into an application developed and simulated in Greenfoot. It is based on an idea similar to ours, namely to adapt existing simulation and visualization tool for "hosting" programs developed by formal methods. On the other hand, they fix the formal method (automata) but not the domain and the tools they provide work in opposite direction: from concrete models (in Greenfoot) to abstract ones (in UPPAAL). It is, however, possible that further development of the experiments with gaming and visualization engines mentioned in the previous section will lead to a creation of a framework similar to that of Balz and Goedicke (2010).

Creation of the tools described here was also inspired by existing solutions for graphical animation and visualization of formally specified systems. There are two such tools for Event-B, available as plug-ins for the Rodin platform. The first is Brama (http://www.brama.fr/), which allows to make custom visualizations by connecting a specification in Event-B with Adobe Flash animation. The connection between specification and animation is defined by a gluing code, written in ActionScript. Brama has been successfully used in industrial projects, for example in the development of a commuter rail platform screen door controller (Lecomte, Servat, & Pouzancre, 2007). The second one is B-Motion Studio (Ladenberger, Bendisposto, & Leuschel, 2009), which tries to be simpler than Brama: animations are composed from pictures and the gluing code is written in B-language. These solutions differ from ours in the fact that visualizations have to be created from scratch but do not have limited domain, they connect with formally developed software on the specification and not implementation level and are limited to one formal method. Their primary purpose is also different: they are intended for presentation of formal specifications to customers and domain experts in an understandable way. Another solution with such purpose is the APEX framework (Silva, Ribeiro, Fernandes, Campos, Harrison, 2010). It connects OpenSimulator (http://opensimulator.org/), a multi-user 3D application server for creation and running of 3D virtual environments, with CPN Tools (http://cpntools.org/). CPN Tools is an editor, simulator and analyzer for Coloured Petri nets (CPN). Despite using a different kind of formal method (CPN) it is similar to ours in utilizing an existing tool (OpenSimulator) to create virtual environments with devices managed by formally developed controllers (here by CPNs). There are also similarities in architecture: CPN Tools and OpenSimulator communicate using TCP/IP via a special communication/execution component, just like Train Director or Open Rails communicate with a control module via TS2JavaConn.

CONCLUSIONS

After two years of using the modified Train Director and TS2JavaConn in our formal methods course we can consider its deployment as a success. When compared to previous years we have seen students more engaged and even students that didn't score very well in other theoretical computer science-based courses managed to accomplish assignments that incorporated the tools without significant problems. One may say that it's not a big deal when everything we get at the end is to watch a simulation, but after coping with such "strange" things like specifying invariants, refinements and conducting proofs students deserve the reward to see the result of their work performing in a believable way. The "believable way" is the key term here. Limited application program interfaces of existing formal methods tools and the time required to perform the "strange" things make it impossible for students to develop within a FM course systems comparable in size and functionality to those they already developed on other courses. And to develop some oversimplified “caricatures” of such systems is hardly motivating and satisfying. The control modules shown here look simple and primitive when compared with source codes of contemporary information systems. But they are something “special”, not an “ordinary” software and deliver exactly what is asked from them, so the simplicity is not perceived as inappropriate.

The practical use of the tools wasn’t without problems. During the first year the need to write specifications of control modules and modify configuration files manually was reported as the greatest setback by students. As a reaction to this we implemented the control module generator to TS2JavaConn. Some students also complained that the process of compilation and running of control modules is somehow complicated. That they have to generate Java code first then compile it and load the compiled module in TS2JavaConn. But it was our intention not to make this a one click task. The reason was to force the students to stay on the formal methods side as long as possible and to run only finished and verified module with corresponding scenario. And not, as during ordinary software development, run the executable program after every minor change in the code.

Another question is the cost of the implementation. Considering the time required for modification of Train Director and development of TS2JavaConn it could be cheaper to develop a new game from scratch. But TS2JavaConn works also with Open Rails and its core is reused in the prototype of the online platform mentioned above. Preparation of various scenarios for Train Director is very cheap; the ones used in the examples above can be created within few minutes. On the other hand, to prepare a scene for a gaming or visualization engine, useable as a virtual application area, requires significantly more time. But here we can employ students of another course that deals with 3d graphics or virtual reality, who can prepare such scenes as their assignments.

The tools and examples presented in this chapter can be downloaded from https://kega2012.fm.kpi.fei.tuke.sk/.

Acknowledgements. This work has been supported by KEGA grant project No. 050TUKE-4/2012: “Application of Virtual Reality Technologies in Teaching Formal Methods”. The authors would like to thank Ján Genči for his suggestions on some titles for the Additional Reading section.

REFERENCES

Abrial, J. R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge: Cambridge University Press.

Abrial, J. R. (2007). Formal methods: Theory becoming practice. *Journal of Universal Computer Science. 13*(5), 619-628.

Abrial, J. R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge: Cambridge University Press.

Almeida, J.B., Frade, M.J., Pinto, J.S., & de Sousa, S. M. (2011). *Rigorous Software Development. An Introduction to Program Verification*. London: Springer-Verlag.

Balz, M., & Goedicke, M. (2010). Teaching Programming with Formal Models in Greenfoot. In: *Proceedings of the 2nd International Conference on Computer Supported Education* (pp. 309-316). Valencia: INSTICC Press.

Boulanger, J.L. (Ed.) (2012). *Formal Methods: Industrial Use from Model to the Code.* London-Hoboken: ISTE – John Wiley & Sons.

Bowen, J. P., & Hinchey, M. G. (2006). Ten Commandments of Formal Methods … Ten Years Later. *IEEE Computer, 39*(1), 40-48.

Cerone, A., Roggenbach, M., Schlingloff, H., Schneider, & G. Shaikh, S. (2013). Teaching Formal Methods for Software Engineering – Ten Principles. In *Proceedings of Fun With Formal Methods, Workshop affiliated with the 25th Int. Conf. on Computer Aided Verification*. Saint Petersburg.

Cristiá, M. (2006). Teaching formal methods in a third world country: what, why and how. In *Proceedings of the conference on Teaching Formal Methods 2006*. London: BCS London Office.

Crocker, D. (2004). Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm In: Proceedings of the *Twelfth Safety-critical Systems Symposium* (pp.19-41) London: Springer-Verlag.

Dijkstra, E.W. (1976). *A Discipline of Programming.* Englewood Cliffs: Prentice Hall.

Feinerer, I., & Gernot, S. (2009). Comparison of tools for teaching formal software verification. *Formal Aspects of Computing, 21*(3), 293-301.

Fitzgerald, J., Larsen, P. G., Mukherjee, P., Plat, N., & Verhoef, M. (2005). *Validated Designs for Object-oriented Systems*. New York: Springer.

Fitzgerald, J., Bicarregui, J., Larsen, P. G., & Woodcock, J. (2013). Industrial Deployment of Formal Methods: Trends and Challenges. In A. Romanovsky & M. Thomas (Ed.) *Industrial Deployment of System Engineering Methods* (pp. 123-143). Berlin – Heidelberg: Springer-Verlag.

Harrison, J. (2008). Theorem Proving for Verification. *Computer Aided Verification, LNCS vol. 5123* (pp. 11-18). Berlin – Heidelberg: Springer-Verlag.

Larsen, P. G., Fitzgerald, J., & Riddle, S. (2009). Practice-oriented courses in formal methods using VDM++. *Formal Aspects of Computing. 21*(3), 245-257

Korečko, Š., Dancák, M. (2011). Some Aspects of BKPI B Language Compiler Design, *Egyptian Computer Science Journal, 35*(3), 33-43.

Korečko, Š., Sorád, J., & Sobota, B. (2011). An External Control for Railway Traffic Simulation, In *Proceedings of the Second International Conference on Computer Modelling and Simulation* (pp. 68-75), Brno University of Technology.

Ladenberger, L. , Bendisposto, J., & Leuschel, M. (2009). Visualising Event-B Models with B-Motion Studio, In *Proceedings of Formal Methods for Industrial Critical Systems, LNCS vol.5825* (pp. 202-204). Berlin – Heidelberg: Springer-Verlag.

Le Lann, G. (1997). An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective. In *Proceedings of the 1997 International Workshop on Engineering of Computer-Based Systems*. IEEE.

Lecomte, T., Servat, T., & Pouzancre, G. (2007). Formal Methods in Safety-Critical Railway Systems. In *Proceedings of 10th Brasilian Symposium on Formal Methods*. Ouro Preto.

Lecomte, T. (2009). Applying a Formal Method in Industry: A 15-Year Trajectory. In *Proceedings of Formal Methods for Industrial Critical Systems, LNCS vol.5825* (pp. 26-34). Berlin – Heidelberg: Springer-Verlag

Leuschel, M., & Butler, M. (2003). ProB: A model checker for B. In *Proceedings of FME 2003: Formal Methods, LNCS vol.2805* (pp.855–874). Berlin – Heidelberg: Springer-Verlag.

Liu, S., Takahashi, K., Hayashi, T., & Nakayama, T. (2009). Teaching Formal Methods in the context of Software Engineering. *ACM SIGCSE Bulletin, 41*(2), 17-23.

Reed, J. N., & Sinclair, J. E. (2004). Motivating study of Formal Methods in the classroom. In Dean, C. N., Boute, R. T. (eds.) *TFM 2004, LNCS, vol. 3294* (pp. 32-46). Berlin – Heidelberg: Springer-Verlag.

Silva, J. L., Ribeiro, Ó. R., Fernandes, J. M., Campos, J. C., Harrison, M. D. (2010). The APEX Framework: Prototyping of Ubiquitous Environments Based on Petri Nets. In *Human-Centred Software Engineering, LNCS, vol. 6409* (pp. 6-21). Berlin – Heidelberg: Springer-Verlag.

Voisinet, J.C., Tatibouet, B., & Hammad, A. (2002). jBTools: An experimental platform for the formal B method. In *Proceedings of PPPJ'02* (pp. 137-140).

Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Computing Surveys 41*(4), 19:1-19:36.

ADDITIONAL READING

Berger, F., & Muller, W. (2012) Towards an Open Source Game Engine for Teaching and Research. In *Transactions on Edutainment VIII, LNCS vol. 7220* (pp. 68-76). Berlin – Heidelberg: Springer-Verlag.

Bicarregui, J., Hoare, C.A.R., & Woodcock, J. The verified software repository: a step towards the verifying compiler. *Formal Aspects of Computing. 18*(2), 143-151.

Bjorner, D. (2001). On teaching software engineering based on formal techniques - thoughts about and plans for - a different software engineering text book. *Journal of Universal Computer Science*, *7*(8), pp.641-667.

Boerger, E. (2003). Teaching ASMs to practice-oriented students with limited mathematical background. In *Proceedings of Teaching Formal Methods 2003* (pp.5-12). Oxford Brookes University.

Brakman, H., Driessen, V., Kavuma, J., Bijvank, L. N., & Vermolen, S. (2006). Supporting Formal Method Teaching with Real-Life Protocols. In *Formal Methods in the Teaching Lab – A Workshop at the Formal Methods 2006 Symposium* (pp. 59–67).

Burgess, C. J. (1995). *The role of Formal Methods in Software Engineering education and industry.* University of Bristol, UK.

Cai, Y., & Goei, S. L. (Eds.) (2014). *Simulations, Serious Games and Their Applications.* Springer Science+Business Media Singapore.

Catano, N., & Rueda, C. (2009). Teaching Formal Methods for the Unconquered Territory. In *Proceedings of Teaching Formal Methods 2009, LNCS, vol. 5846* (pp. 2-19). Berlin – Heidelberg: Springer-Verlag.

Freitas, S., & Liarokapis, F. (2011). Serious Games: A New Paradigm for Education? In *Serious Games and Edutainment Applications*. London: Springer-Verlag.

Forišek, M., & Steinová, M. (2013). *Explaining Algorithms Using Metaphors*. Springer.

Gibson, J. P., & Méry, D. (1998). Teaching formal methods: lessons to learn. In *Proceedings of the 2nd Irish conference on Formal Methods* (pp. 56-68).

Gibson, J. P. (2008). Formal methods-never too young to start. In Z. Istenes (Ed.), *Proceedings of Formal Methods in Computer Science Education* (pp.151-160), Budapest.

Gibson, J. P. (2008). Weaving a Formal Methods Education with Problem-Based Learning. In *Proceedings of the Third International Symposium ISoLA 2008, CCIS vol.17* (pp. 460-472). Berlin – Heidelberg: Springer-Verlag.

Glass, R. L. (2004). The mystery of formal methods disuse. *Communications of the ACM*, *47*(8), 15-17.

Heitmeyer, C. (1998). On the need for practical Formal Methods. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS vol. 1486* (pp. 18-26). Berlin – Heidelberg: Springer-Verlag.

Ishikawa, F., Taguchi, T., Yoshioka, N., & Honiden, S. (2009) What Top-Level Software Engineers Tackle after Learning Formal Methods: Experiences from the Top SE Project. In *Proceedings of Teaching Formal Methods 2009, LNCS, vol. 5846* (pp. 57-71). Berlin – Heidelberg: Springer-Verlag.

Jaspan, C., Keeling, M., Maccherone, L., Zenarosa, G. L. & Shaw, M. (2009) Software Mythbusters Explore Formal Methods. *IEEE Software, 26*(6). 60–63.

Malan, D., & Leitner, H. (2007). Scratch for budding computer scientists. *ACM SIGCSE Bulletin 39*(1), 223–227.

Moller, F., & O'Reilly, L. Formal Methods for First Years. In *Proceedings of Fun With Formal Methods, Workshop affiliated with the 25th Int. Conf. on Computer Aided Verification*. Saint Petersburg.

Morrison, B. B., & DiSalvo, B. (2014). Khan academy gamifies computer science. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 39-44). ACM.

Race, P. (1998). Teaching: Creating a Thirst for learning? In Brown, S., Armstrong, S., & Thompson, G. (Eds.) *Motivating Students* (pp. 47-57). London: Kogan Page.

Spies, K., & Schatz, B. (2006). A Playful Approach to Formal Models — A field report on teaching modeling fundamentals at middle school. In *Formal Methods in the Teaching Lab – A Workshop at the Formal Methods 2006 Symposium* (pp. 45–52).

Trow, M. (1973). *Problems in the Transition from Elite to Mass Higher Education.* Carnegie Commission on Higher Education. Berkeley,California.

Whitton, N. (2010). *Learning with Digital Games. A Practical Guide to Engaging Students in Higher Education*. New York: Routledge.

Woolsey, K. (2008). Where is the New Learning, In R.N. Katz (Ed.), *The Tower and the Cloud:Higher Education in the Age of Cloud Computing* (pp. 212-218). Educase.

KEY TERMS AND DEFINITIONS

Control module: A program controlling some device or devices (e.g. signals and switches in a track layout). It is a reactive system, i.e. it reacts to external events such as requests from trains (definition specific to this chapter).

Formal method: Mathematically based technique for specification, analysis, development and verification of computer systems. Consists of a formal language and procedures that allow to perform desired tasks with specifications written in the language.

Formal specification: A description of a (hardware or software) system in a formal language.

Formal language: A language with unambiguously defined syntax and semantics. The semantics is defined using some mathematical apparatus.

Formal verification: A process of using some mathematical apparatus to prove that a system has corresponding formally specified properties.

Railway scenario: A scenario for a railway simulation game, consisting of a track layout and a schedule for trains that operate on the layout (definition specific to this chapter).

Refinement: A transformation from an abstract specification of a system to a concrete, executable one.

Track device: A signal, a switch or a track section in a track layout (definition specific to this chapter).