# A Toolset for Support of Teaching Formal Software Development

Štefan Korečko, Ján Sorád, Zuzana Dudláková, and Branislav Sobota

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 041 20 Košice, Slovakia
{stefan.korecko,zuzana.dudlakova,branislav.sobota}@tuke.sk,
jansorad@gmail.com

**Abstract.** Teachers of formal methods courses often experience disinterest or even disgust towards the topic from software engineering students. As one of the significant reasons of this situation we see the fact that students are not in touch with domains where their use is desired and worth the effort. In this paper we deal with a toolset we developed to improve the situation. The toolset brings to students, in a virtual form, one of the most successful domains of formal methods application - railway systems. It consists of a modified version of a railway centralized traffic control simulator called Train Director and a tool that allows signals and switches in a railway scenario, simulated by Train Director, to be controlled by a separate formally developed control program. We briefly describe the toolset and its typical use within a formal methods course and discuss its usability with respect to various formal methods.

**Keywords:** formal methods, teaching, software development, railway systems, virtual laboratory.

## 1 Introduction

Almost every university teacher who dedicated a portion of his career to introducing software engineering students to the world of formal methods (FM), especially to those heavy weighted ones that involve formal verification and refinement, faced several serious problems. And in the era we live in, the era of massification of higher education, one of the most important problems is how to motivate students to enrol into formal methods courses and to stay in them. Of course, we often get the "Why should I learn this language (method, approach, etc.)?" question in "normal" software engineering subjects as well. But there it can be easily answered by pointing out a number of (local) companies whose employees use them on regular basis. However, in FM courses we get a more serious questions, like "Why on earth should I deal with such terrifying stuff like formal semantics or, heaven forbid, mathematical proofs?". And the easy answer is not here. There are only few companies worldwide that use FM in software development and they are almost exclusively located in the most developed countries.

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 278–283, 2014.
© Springer International Publishing Switzerland 2014

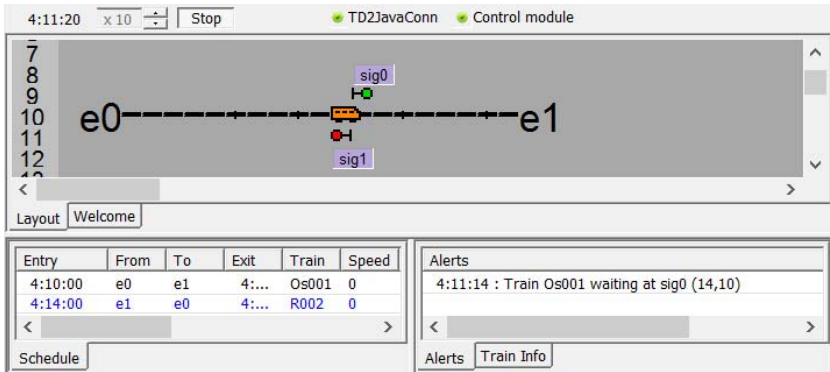
Instead of it we can, together with the authors of [3], focus on a noble goal of educating rigorous software engineers who will change the way software is produced. But will our students follow? It is our strong belief that they will if we let them play with FM in an appropriate setting. We agree with Almeida et al. [1] that FM should not be used everywhere, but primarily in cases when reliability, safety or security are a concern. We can hardly persuade students to put an extra effort to formal verification of some typical information system, a computer game or a mobile phone firmware when they already know that such software is usually released with several bugs, fixed by updates afterwards. And nothing really bad ever happens. The problem is that these are exactly the types of systems our students encounter during their university study. So, we need to introduce them to a domain where use of FM is appropriate, where they will feel the need of that extra effort. And we have to let them develop something for that domain, using FM. But what domain to choose? We should pick up one where software failures may have tragic consequences. It is also important that most of the students are familiar with the domain and can imagine these consequences affecting their lives. Moreover, there should be real-life cases from the domain where human lives are already under the control of automated systems. There are several candidates but one of the most appealing are railway systems. The fact that this domain is one of few where FM reached a mature level [2] is a nice and important bonus.

The domain has been selected, now the question is how to bring it to the students. The real railway is definitely out of our reach and to let students develop a console application on the basis of some text document form of assignment is hardly motivating. This is exactly where our toolset can help. It replaces the real railway with a virtual one, represented by a simulation game called Train Director (`http://www.backerstreet.com/traindir`). The modifications we made to the game allow it to be connected to a separately developed control program (module), which responds to requests from simulated trains by manipulating switches and signals. And these control modules are the very pieces of software the students develop using formal methods. The control modules are Java applications, so the toolset is usable with any FM for which a Java code generator exists. The appearance and operation of the toolset and control modules, their place in a FM course and adaptability to various FM are described in the next section. The final one deals with teaching experiences, related work and plans for future development.

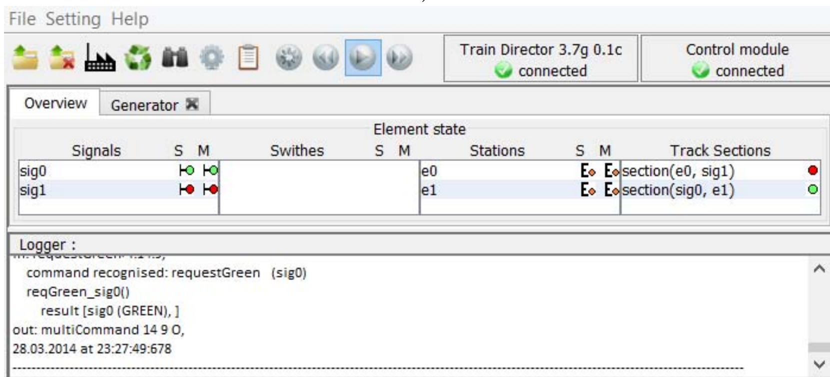
## 2 The Toolset and Its Use

The toolset itself consists of two tools - a modified version of *Train Director*, an open source simulator of the railway centralized traffic control and *TS2JavaConn*, a Java application whose primary role is to provide communication between the simulator and control modules.

**Train Director** is a game that allows a user to create and simulate a railway scenario, which consists of a track layout and a train schedule. The user's task




a)




b)

**Fig. 1.** The toolset during a simulation: Train Director (a) and TS2JavaConn (b)

during the simulation is to throw switches and clear signals in such a way that the trains will follow the schedule. The game has its own logic that prevents collisions and changes some signals automatically. It also provides a simple server interface for an external control. We modified the game by disabling the internal logic and implementing train collisions and naming of signals and switches. We also enhanced the server interface to be able to communicate with TS2JavaConn. The modified Train Director sends messages to TS2JavaConn every time a train stops before a red signal (*requestGreen* message), wants to enter a track layout (*reqestEnter*) or departure from a station (*reqestDepartureStation*). These messages also contain name of the corresponding signal, entry point or station, name of the train and names of following stations the train should visit according to the schedule. A *sectionLeave* message is sent when a train leaves current track section and a *sectionEnter* message when it enters a new one. For the sake of simplicity a track section always starts and ends at some signal, switch or entry point. The modified Train Director can also receive messages. These messages are commands from TS2JavaConn to, for example, start or stop a simulation or

to change the state of a signal or switch. In Fig. 1 a) we can see Train Director during a simulation of simple scenario that consists of two entry points `e0` and `e1` and two signals `sig0` and `sig1`. There are two track sections, `e0_sig1` and `sig0_e1`.

**TS2JavaConn** (Fig. 1 b) was necessary because Train Director is a C++ application and control modules are in Java. We have chosen Java because of its popularity among students and wide support in FM tools. Having TS2JavaConn as a separate tool also allows to easily replace Train Director with another simulator. The tool provides a GUI where a user can load a control module (first button in the toolbar in Fig. 1 b), unload the module ($2^{nd}$ button) open a tab with a module generator ($3^{rd}$ button), reset the connection with the simulator ($4^{th}$ button) or remotely control the simulation in Train Director (round buttons). In the "Element state" part of the "Overview" tab a user can observe in which state track elements are in the simulator (S) and in the module (M). Communication between the tools can be watched in the "Logger" part. For each scenario the current version of the module generator can create a control module template in Java and in specification languages of formal methods B-Method and Perfect Developer, together with a corresponding configuration file.

**The control module** itself is a Java application where one "main" class contains methods that react to the messages from Train Director and variables that represent devices from the controlled scenario. How exactly these methods and variables are mapped to the messages from and devices in Train Director is defined in a mandatory text-based *configuration file*. The possibilities are wide: for data representation 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 a) we need two non-parametric methods (`reqGreen_sig0` and `reqGreen_sig1`) or one parametric method (`reqGreen(sig)`) to handle the requestGreen messages for `sig0` and `sig1`. The number of additional classes and libraries in control modules is not limited, so the modules can be really sophisticated and complex applications. One may ask why we bother with the non-parametric representation, but our experience shows that more complex data representation, necessary for the parametric one, usually makes an automated verification in FM tools impossible even for very simple scenarios.

**The communication** between the simulator and a control module can be seen in Fig. 1, which shows the tools exactly after the moment when a request for clearing the signal `sig0` is received from the train `Os001`. As we can see in the "Logger" part, TS2JavaConn responds by calling `reqGreen_sig0` method from the connected control module. The method changes the value of a variable that represents `sig0` and this change is sent back to the simulator (as so-called *multiCommand* message) where it sets `sig0` to green.

**During a FM course** the toolset is useful in both lectures and practices. In lectures the whole method taught can be illustrated by examples utilizing the toolset. Even very simple scenarios and control modules are able to demonstrate

benefits (e.g. we can specify safety conditions and prove that they hold in every state of given program) and drawbacks (e.g. we cannot prove that the safety conditions we specified are the right ones) of FM. Advanced concepts can be explained as well. For example, on a refinement from a data representation that corresponds 1:1 to devices in given scenario to a more effective one or on a control module composed from reusable components for individual track types (a straight track, various junctions, etc.). On practices the toolset primarily provides a virtual laboratory environment and is usually used in the following way: First a teacher creates a new scenario, or modifies an existing one, in Train Director. When creating it he should restrict himself to simple red/green signals and two-way switches. Then he presents the scenario to students with a task to create a dependable controller for it. The students can then play with the scenario in Train Director: if it is disconnected from TS2JavaConn, switches and signals can be operated manually. After getting familiar with the scenario the students generate an empty module using TS2JavaConn and start to develop the controller itself, using given formal method and its tools. Finally, they return to the toolset and run the finished and compiled module with the scenario.

**Adaptability** of the toolset to various FM was one of the primary concerns during its development. It has been achieved by making the toolset as independent from actual FM tools as possible. There are only two "common points". The first one is when a new control module template is generated. Only languages of two FM are supported yet, but the module generator is based on the Apache Velocity template engine, so new ones can be added easily. The second point is when a finished module is connected to and run with the corresponding scenario. Here the only requirement is that the module has to be a Java application with an appropriate interface. And Java code generators are available for a wide variety of FM tools. For example, Perfect Developer (`http://www.eschertech.com`) and VDM++ Toolkit (`http://www.vdmtools.jp`) have built-in generators, for the Rodin tool (`http://www.event-b.org`) of the Event-B method they exist in a form of plug-ins (e.g. EB2J, `http://eb2all.loria.fr`) and for B-Method we provide our own generator, called BKPI compiler, optimized for the use with the Atelier-B tool (`http://www.atelierb.eu`). We had tested the toolset with all these methods and code generators and only in the case of EJ2B it was necessary to alter the generated Java code (an explicit constructor was added). In all other cases it was enough to modify configuration files of the modules.

## 3 Conclusions

The toolset was already used during two runs of a FM course, which teaches Petri nets and B-Method, at the home institution of the authors. Petri nets were explained with abstract models of synchronization problems while almost all examples in the B-Method part were prepared and presented using the toolset. On practices the toolset was used in the way described in section 2. Based on the students' feedback we can conclude that our belief has been confirmed. When compared to previous years the students were more engaged and even those who

didn't score very well in other theoretical computer science-based subjects managed to accomplish assignments that incorporated the toolset without significant problems. Some of them reported that they enjoyed the B-Method part more that the Petri nets part despite its significantly higher difficulty. And all of them enjoyed the moment when they have finally seen their control modules running in the toolset. Of course, problems were reported, too. After the first run (in 2013) the need to write configuration files and whole control modules manually was identified as the greatest setback. To improve the situation the module generator of TS2JavaConn has been implemented. Students also reported that it takes too long to get the modules from FM side to the toolset. But this was intentional, to prevent students from using the "modify-compile-run" cycle too often.

The belief we presented is also supported by other educators. The work [4] shares our view on importance of motivation with respect to massification of higher education and points out that students will see little benefit in developing ordinary systems using FM. The authors of [2] see the importance of an appropriate experimentation platform, which is exactly what our toolset tries to establish, as high enough to make it one of their ten principles (no. 6).

The future development of the toolset will focus on an improvement of the module generator and replacement of Train Director by 3D train simulator Open Rails (`www.openrails.org`).

This paper dealt primarily with the actual version of the toolset and its use in a FM course. Additional information, such as the general idea behind the toolset, related work that led us to the idea, examples of control modules and the reasons why Train Director was chosen, can be found in other papers by the authors. These papers, the toolset, the BKPI compiler and a set of examples with control modules developed using various FM can be downloaded from `https://kega2012.fm.kpi.fei.tuke.sk`.

**Acknowledgments.** This work has been supported by KEGA grant project No. 050TUKE-4/2012: "Application of Virtual Reality Technologies in Teaching Formal Methods".

## References

1. Almeida, J.B., Frade, M.J., Pinto, J.S., Melo de Sousa, S.: Rigorous Software Development. An Introduction to Program Verification. Springer, London (2011)
2. Cerone, A., Roggenbach, M., Schlingloff, H., Schneider, G., Shaikh, S.: Teaching Formal Methods for Software Engineering Ten Principles. In: Fun With Formal Methods, Workshop Affiliated with the 25th Int. Conf. CAV 2013 (2013)
3. Cristi, M.: Teaching formal methods in a third world country: what, why and how. In: Proceedings of the 2006 Conference on Teaching Formal Methods: Practice and Experience (2006)
4. 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)