USING FORMAL METHODS IN SOFTWARE ENGINEERING EDUCATION

Anastas Misev
University of Sts. Cyril and Methodius
Faculty of Natural Sciences and Mathematics

Boro Jakimovski
University of Sts. Cyril and Methodius
Faculty of Natural Sciences and Mathematics

ABSTRACT

Formal methods have wide usage in software engineering. One especially important category of software engineering where formal methods are irreplaceable is software engineering for critical systems. Critical systems represent software products that require high level of correctness in requirements specification, software design and the final product. Using different formal methods we can help in development of safer and more reliable critical systems. In this paper we give an overview of the formal methods and tool used in Software Engineering. We also present our experience in lecturing a course in SE for CS at a post graduate level, the methodology used in to connect the theory with practice by presenting several formal methods. Out final goal was the use of temporal logic in SE. We also describe the tools used and show several case studies presented to the students and the lessons learned during the course.

I. INTRODUCTION

Software engineering methods used in specification and design are usually informal and can contain contradictions and ambiguities. This later can lead to faulty design and at the end a faulty implementation. This problem is especially important in when developing software for critical systems. Critical Systems (CSs) are those whose operation poses a risk to human life, health, economy or environment. Typically, CSs are large and complex industrial systems or products which have been constructed through the effort of multi-disciplinary teams. CSs represent software products that require high level of correctness in requirements specification, software design and final product. Using different formal methods we can help in development of safer and more reliable critical systems.

In this paper we give an overview of the formal methods and tool used in Software Engineering. We also present our experience in lecturing a course in SE for CS at a post graduate level, the methodology used in to connect the theory with practice by presenting several formal methods. Out final goal was the use of temporal logic in SE. We also describe the tools used and show several case studies presented to the students and the lessons learned during the course. In chapter 2 we will cover most widely used formal methods in SE. Chapter 3 will present the outline the formal methods in the SE for Critical Systems course for Joint Master Studies in SE. Following this in chapter 4 we conclude with our view of the course and lessons learned for the future years.

II. FORMAL METHODS USED IN SE

A. Formal specification

Formal methods are most frequently used in the requirements analysis and high-level design. A formal specification is a characterization of a planned or existing system represented in a formal language. The main reason for using formal languages is their clear expressive power which allows for non-ambiguity and consistency in the specification.

There are several ways to increase the certainty that a specification expresses the intentions of its author, and that what it says is true. This aspects include:

- parsing – allow for consistent reading of the specification every time with focus on the syntactic correctness
- type checking – consistent usage of specified functions (semantic correctness)
- executing all or part of the specification – allows for exploring and debugging of the specification and developing and evaluating of test cases.
- checking weather definitions or theorems entailed by the specifications are well formed – meaning that the specification can be proven to be consistent
- demonstrating consistency for axiomatic specifications – this is usually a property of the chosen formalism that strengthens its grounds as being formally correct and consistent.

Many formal specification languages extend the underlying pure logic with programming language concepts and constructs such as type systems, encapsulation, and parameterization. This leads to increased expressiveness of the formal language while retaining the precise semantics of the underlying logic. Even though a program can be viewed as a specification, a specification is typically not a program and often contains such non computational elements as high-level constructs and logical elements (e.g., quantifiers). The basic difference is that a program specifies completely how something is to be computed, whereas a specification expresses constraints on what is to be computed. As a result, a specification may be partial or “incomplete" and still be meaningful, but an incomplete program is generally not executable. [1].

Lamport [2] identifies several functions of the formal specification. One could be “a contract between the user of a system and its implementer. The contract should tell the user everything he must know to use the system, and it should tell the implementer everything he must know about the system to implement it. In principle, once this contract has been agreed upon, the user and the implementer have no need for further communication.”. He identifies three issues:

- First is that the most important functions of a formal specification is analytic - using the deductive apparatus of the underlying formal system, a formal specification serves as the basis for calculating, predicting, and (in the case of executable specifications) testing system behaviour. However, a formal specification may also serve an important descriptive function, that is, provide a basis for documenting, communicating, or prototyping the behaviour and properties of a system.
- Second, a (completed) specification represents the formalization of a consensus about the behaviour and properties of a system.
- Third, while in principle, analyzed contract precludes the need for further communication among the interested parties, in practice, moving from informal requirements to a formal specification and high-level design is an iterative rather than a linear process; issues exposed in the development of the formal specification may need to be factored back into the requirements, and similarly, issues raised by the high-level design may percolate back to impact either the formal specification, the requirements, or both.

B. Formal analysis

Formal analysis refers to a broad range of tool-based techniques that can be used to explore, debug, and verify formal specifications, and to predict, calculate, and refine the behaviour of the specified systems.

There are three categories of formal analysis techniques:

- Automated deduction or theorem proving
- Finite state methods
- Direct execution, simulation and animation

Automated deduction or theorem proving refers to the automation of deductive reasoning in formal specification methods that support such apparatus. Deductive methods are especially useful when reasoning about infinite-state systems. They are typically preferred for abstract, high-level specifications and data-oriented applications.

Finite-state methods refer to techniques for the automatic verification systems with finite states and also some infinite-state systems can be reduced by some tools to finite-state equivalents. Given a formula specifying a desired system property, these methods determine its truth or falsity in a specific finite model. Some of most commonly used finite-state methods are:

- Temporal logic
- LTL (Linear-time Temporal Logic)
- Branching Time Temporal Logic
- μ-Calculus

The final category of formal analysis techniques is the direct execution, simulation and animation which are used to observe the behaviour of the model of a system. This aspect of formal analysis is the least formal technique which simulates the formal specification of the system. The reason for using observational techniques is because typically, models used for verification cannot expose their own inaccuracy and, conversely, models used for conventional simulation cannot confirm their own correctness [3]. This is very important in development of system specification since it assures that the system is specified according to the requirements, in other words it allows the specification and underlying model to be "debugged" relatively early in the SE life cycle.

III. COURSE ORGANIZATION

The postgraduate course named Software Engineering for Critical Systems aims to introduce and critically analyse CSs. It explores the requirements for the engineering of CSs and the role of formal approaches in their life cycle. One of the important learning outcome of the course is for the students to critically evaluate the use of formal methods in the life cycle of CSs and the use of temporal logic for the engineering and re-engineering of CSs.

An important aspect in the development of a CSs is how to cope with their "evolution" [4]. The evolution of a software system could be due to several factors such as changes in the original requirements, adopting a different hardware platform or to improve its efficiency. These kind of changes become more emphasised when developing time-critical systems. Because of their complexity, the likelihood of subtle errors is much greater and some of these errors could have catastrophic consequences such as loss of life, money, time or damage to the environment.

The structure of the course is such that it emphasises the combination of theoretical aspects (like formal methods) with practical issues of software engineering. It is an example of how theory can be applied in practice, a feature that many theoretical courses omit. In this paper we give a focus on the tutorials and labs that are accompanying the course which covered the formal analysis of SE for CSs.

The course tutorials final goal is the usage of Interval Temporal Logic (ITL)[5] as finite-state formal analysis method because of the existence of its executable subset Tempura. The tutorials are structured with an intention to gradual introduce several formalisms that students already know and try to connect them to formal analysis aspects of SE. Thus the first several tutorial topics include:

- First Order Logic – covering the rationale, syntax, semantics, usage and knowledge engineering using first order logic;
- Logics and Programming – covering the notion of function of a program and Hoare logic;
- Introduction to Temporal Logic – covering basic principals of temporal logic and elementary syntax of TL.

After introducing the connection between logics and programming the tutorials focus on ITL as the formal analysis method used in this course. The topic include:

- Introduction to ITL – covers the semantic model of ITL, transition from informal to formal specification using ITL;
- ITL Syntax – covers the syntax of non-temporal ITL expressions and formulas, satisfiability and validity of ITL formulas, temporal ITL expressions and formulas;
- Derived ITL formulas – covers the more complex temporal formulas.

The final part of the tutorials is to animate the formal specification using the executable subset of ITL named Tempura[5, 6]. Tempura is a tool maintained by STRL[7] that enables animation of some ITL formulas into an corresponding sequence of states named an interval, that satisfies the needed properties.

Tempura offers a means for rapidly developing, testing and analysing suitable ITL specifications. As with ITL, Tempura can be extended to contain most imperative programming features and yet retain its distinct temporal feel. The use of ITL, together with its subset of Tempura, offers the benefits of traditional proof methods balanced with the speed and convenience of computer-based testing through execution and simulation [6].

The tutorials covering Tempura are party hands-on lab exercises covering the following topics:

- Tempura[6] – covering the executable ITL subset, tempura interpreter;
- Anatempura – covering the extension of Tempura tool that supports validation and verification of the software system in the form of simulation and run-time testing in conjunction with its formal specification.

The coursework given to the students includes several exercises introduced during the tutorials and final project. We start with simple assignments, mainly modification of the examples. Gradually, we move to more complicated examples, using real life systems. In we show a list of some exercises given to the students.

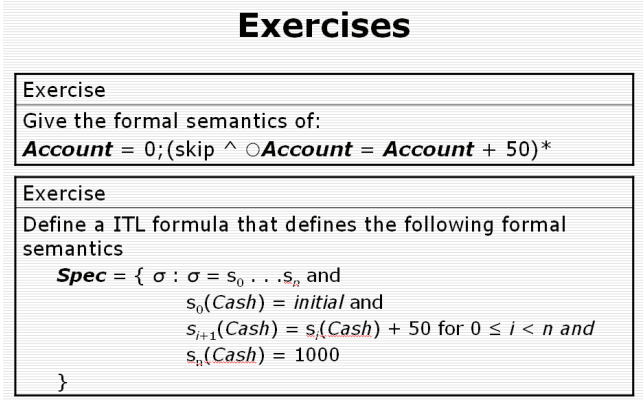

Fig. 1. Example exercise from the course

Final coursework project aim is to help the students develop an understanding of safety-critical system. For the past cycle, the objective of the coursework was to develop a software simulation of an insulin pump. The specification of the coursework was given in textual form as well as partially in Z specification.

The deliverables of the coursework included:

- ITL specification of the system
- UML design of the software
- Java source code of the insulin pump software
- User manual
- Description of the algorithm
- Anatempura test cases that interact with the Java code and execute different usage scenarios

IV. Conclusion

Considering that this course was an elective, not many students attended it. The main reason was that it was partly a theoretical course. Even though the students that attended the course found it to be a great course, with much applicability and benefits. They also appreciated the connection between the theory and practice, because in their previous education the theory oriented courses lacked applicability.

At the end of the course, judging from the coursework results it turned out that ITL was the hardest part of the course work. Only 40% of the students had manifested satisfactory level of ITL knowledge. This indicates that it can be quite difficult for the students to understand formal methods. To improve this, more case studies should be demonstrated to them, especially from real life systems.

During the tutorials have concluded several key points:

- Tutorials should be based on student interactions since theoretical tutorials lead to lost of attention among students.
- Theoretical concepts should be explained through practical usage;
- Even highly complex theory such as ITL can be successfully comprehended by students if appropriate real-life case studies are presented.
- Usage of tools for animation of formal specification significantly alleviates the process of understanding of theoretical concepts to the students.

We will include all of these conclusions into the future tutorials of the mentioned course. We hope that we can increase the level of ITL knowledge and even more demonstrate that this and other theoretical concepts can have great applicability in practice.

V. References

1. NASA, Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion [NASA-GB-001-97], (1997).
2. L. Lamport. A Simple Approach to Specifying Concurrent Systems. Communications of the ACM, p32, (1989).
3. C. Landauer. Discrete Event Systems in Rewriting Logic. In J. Meseguer, editor, First International Workshop on Rewriting Logic and its Applications, p309, Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4. (1996)
4. S. Zhou, H. Zedan, A. Cau, Run-time Analysis of Time-critical Systems, Appeared in Journal of System Architecture, Elsevier B.V., volume 51, number 5, p. 331, (2005)
5. The ITL homepage. URL http://www.cse.dmu.ac.uk/STRL/ITL//index.html
6. B. Moszkowski, Executing Temporal Logic Programs, Cambridge University Press, Cambridge UK, (1986).
7. The Software Technology Research Laboratory (STRL) homepage. URL http://www.cse.dmu.ac.uk/STRL/