Integrating Formal Methods into Computer Science Curricula at a University of Applied Science

Paul Tavolato[1] and Friedrich Vogt[2]

[1] University of Applied Sciences, St. Pölten, Austria
[2] Visiting Professor at University of Applied Sciences, St. Pölten, Austria

July 22, 2012

Abstract. This paper discusses the topic of integrating formal methods in computer science curricula at universities of applied sciences on a general scale: Why is the teaching of formal methods within such curricula essential? And how could we do that?
The main points are: Formal methods bring essential improvements to the daily practice of software development. Alumni from universities of applied sciences are the most important messengers in this field. Motivating these students to realize the importance of formal methods for real world problems is therefore an important educational challenge.
Our approach to this challenge comprises three different courses in a master program: Theoretical Foundations of Computer Science, Software Engineering and Model Checking. It comprises ideas with regard to contents as well as ideas with regard to didactic questions. We are currently on the way to implement these ideas within a master program in Information Security at the University of Applied Sciences at St. Pölten, Austria where Model Checking is grounded on the TLA+ Language and the recent version of the Toolbox.

1 Introduction

At universities of applied sciences there are usually limiting factors regarding the didactic concept one has to choose for teaching formal methods:

- Students have only very limited theoretical background from their bachelor program (not very much math, nearly no computer science theory).
- The curriculum is very job oriented.
- Students are strongly focused on the direct applicability of the knowledge they are taught.

On the other side why do we want to teach formal methods in such an environment? Computer science as a subject is characterized by rapid changes in nearly every aspect: Things you learn today are doomed to be obsolete tomorrow. The only exceptions from this terribly speedy trend are the theoretical foundations of computer science. So there is a great need for a sound scientific background which enables the students to cope with future challenges. The better this scientific and theoretical background the easier the students will be able to learn new versions of the technology coming up every year. We are responsible to equip our students not only with knowledge and skills readily applicable in a vocational environment, but also to enable them to cope with future requirements. Model checking and hence the mastering of formal methods like TLA+ [4] is a technology at the fringe of becoming a practical skill and is therefore a very good example for what we mean.

Our goal in this educational project was the integration of formal methods into a curriculum that is mainly based on the applicability of the skills and the knowledge taught. This includes teaching the fundamentals of theoretical computer science, showing how formal methods could be used in software engineering, teaching formal specification methods such as TLA+ [4], and showing the way how such formal methods can be applied to practical problems and how this knowledge gives a deeper understanding of the subject and therefore will help to anticipate future directions in computer science.

2 Requirements for teaching formal methods

Taking into account the before mentioned constraints we found the following critical factors that we must solve in order to achieve our goals:

- We must teach at least some fundamentals of theoretical computer science to enable the students to deal with formal methods. This includes mainly modeling skills using various formal possibilities (say formal languages, logic, mathematical domains such as graph theory, and similar things).
- The practical applicability of the theory taught must be understandable to students. This means we must not drift away too far from everyday problems which the students will accept as being a vital part of their future working environment.
- Instruction must be problem oriented. This is a consequence from the last requirement: Why not attach the formal methods to real problems and so show in advance that the acquisition of theoretical knowledge is a necessary must in order to be able to solve the problem.
- Motivation is a key factor in any teaching process. If the students do not know why they have to learn something, they will not learn it (or only acquire a minimum of semi-useless temporary knowledge just sufficient to pass the exam). And compulsory measures such as tough exams with serious consequences of failures are not achieving the desired effects. The philosopher Karl Popper brought it to the point when saying "Answers to questions not asked cannot be understood".

Satisfying all these requirements will only be possible in an ideal situation; but in practice we often have to deal with situations that are restricted by various limitations such as time constraints. In our special situations we additionally have to deal with the strong focus on security coming from the overall goal of the study course. Choosing appropriate examples inevitably leads into the discussion whether to limit ourselves to simple toy examples (which make explanations easier but are much more difficult for motivation) or to try real world examples (which have a strong positive potential on motivation but are usually voluminous and therefore harder to understand).

3 Proposal for an integration of formal methods

We will now describe how we implement these ideas in a real setting. We are teaching in a master course "Information Security" at the University of Applied Science in St. Pölten, Austria. Students in this master course usually have a bachelor in IT Security or some other computer science related program.

We define our teaching goals as follows:

- Students should acquire a fundamental understanding of the usefulness of formal methods as a basis for practical work.
- Students should be able to explore relevant theoretical material on their own.
- Students should be able to apply formal methods such as (TLA+) to real world problems.

3.1 Contents

3 courses within the first 2 semesters of the master program are coordinated tightly to achieve the goals:

- Theoretical Foundations of Computer Science,
- Software Engineering,
- Model Checking and Testing.

The course "Theoretical Foundations of Computer Science" focuses on modeling and tries to convince students that technology as well as science is based on the building of some kind of model – and that you need some kind of language for describing these models. Especially in computer science these languages are required to be formal. One of the examples for such a language taught in this course is logic. Propositional calculus is only touched upon as students are well acquainted with it; predicate calculus is done in more depth; one of the main ideas is to communicate to the students that there are different kinds of logic, such as tri-valued logic (which they should know from SQL), fuzzy logic or temporal logic. Another main focus of this course is the distinction between syntax, semantics and pragmatics of a language. So this course lays the ground for using formal methods.

Further topics include the classical area of formal languages and automata with the Chomsky hierarchy of languages and the corresponding automata; elementary results about (un)decidability and the complexity of algorithms.

The course “Software Engineering” gives a rather standard view of the field, using UML notation for the analysis and design of systems. The course starts out with an overview of process models for software development (waterfall life cycle, prototyping, Unified Process, agile methods and Scrum). Project planning and project management are not treated as there is a special course on project management in the curriculum. The next chapter focuses on systems analysis and gives a first introduction to UML as a tool to describe the results of the analysis. This is followed by a large chapter on systems design covering architectural design as well as detailed design and going deeper into UML. Here we also give a short introduction to OCL to show how more formal definitions of requirements and system behavior could be integrated into a design.

The lecture on Model Checking is designed as a block lecture with six three hour blocks within two weeks. On the first day a general survey is given including a first look and feel of the features of the TLA+ Toolbox [1] explaining a simple example as e.g. the OneBitClock. The content of each of the following block lectures consists of two parts. The first part consists of an introduction to logic with the emphasis of natural deduction for propositional and predicate logic, temporal logic and some discussion of the possibility to specify liveness properties in TLA+ [4]. The second part at all five remaining block lectures emphasizes the TLA+ Toolbox [1] by explaining several examples like HourClock (more on TLA+), Fowler (translating a FSM description into a TLA+ description), PurchaseOrder (emphasizing message exchange in a simple protocol) and CarTalkPuzzle vs. WeightSelect to emphasize the difference between Behavior and No-Behavior specifications.

3.2 Didactical and methodological considerations

A special framework condition of our curriculum is that within the mentioned programs all instruction is organized by topics, which means that there is only one subject taught at a time (for a period of say 2 or 3 weeks). Subjects are assigned lecture hours (say 45 hours, which would be equivalent to a 3-hour-per-week lecture for a 15-week semester) and an estimated amount of time for self-study (in the example above that might be 75 hours). So put together 45 + 75 = 120 that amounts to a 3 weeks period assuming that students’ workload is a 40-hours week. Of course this is an average value varying from student to student (as we all know some students are smarter than others. . .). The number of students in the courses is 20. This concept has been implemented on an experimental basis last year accompanied by a scientific evaluation. The consistently positive results of this evaluation [2] led to the decision to implement this framework throughout the whole study program starting with the fall semester 2012.

3.3 Theoretical Foundations of Computer Science

The course is scheduled for 9 workdays in a row with an extra day for the final exam afterwards.

The course starts with an overview of modeling concentrating on scientific models and on languages used to describe models. Main points here are the aspects of syntax, semantics and pragmatics as defining constituents of a language. Simple examples shall give an introduction to the road of formal modeling: Starting with simple problems different possibilities of formal models to express these problems are discussed. Hand in hand with it goes the choice of a language to describe the model, including the difference of syntax and semantics of the description.

After this introduction a set of problems/projects is explained and groups of 4-5 students are assigned to each problem. The problems are very different and reflect modeling tasks in different areas of computer science with an emphasis on information security. Examples for such tasks are:

- Complexity of specific algorithms (such as algorithms to crack passwords or various encryption methods)
- Formal language theory with special applications to vulnerabilities of software interfaces (such as described in [5])
- Modeling of security constraints with logic constructs (such as defining protocols by logic formulae [3])

(Note that model checking is not part of the problems as a special lecture course for this topic is available.)

This happens on the first day of the course. On the second day of the course the students must specify their problem precisely and develop a plan about what knowledge is necessary to solve the problem and where to get the information from. This work is guided by the lecturer who moves between the groups, gives hints and provides learning material. At the end of the day every group should have a clear plan about their work for the next days.

The next 2 days are devoted to autonomous work of the students. They gather the information needed and try to apply it to their problem.

The next 2 days are again tutored work: The lecturer helps the students to complete the problem solution and to prepare a presentation of the solution and the underlying formal methods. These presentations are given the next day.

When working on the tasks the students must engage in the corresponding theoretic topic. In case of the complexity task they have to acquire information about computational models and models of algorithms and try to find a reasonable measure (complexity classes); in case of the formal language task they have to engage in syntax theory and the Chomsky hierarchy of languages, the fundamentals of automata including the notion of Turing machines; the constraint modeling forces the students to work with the fundamentals of logic.

So each group has to acquire knowledge in a different field of formal modeling. Note that the emphasis is not on the knowledge acquired but on the way how to acquire knowledge. Of course this will not lead to a thorough treatment of the subject – but it will give an idea on how to apply formal methods to a problem in step with vocational practice. The main teaching problem here is to find a balance between the necessary amount of fundamentals and a naive use of formal methods. Too much of the fundamentals will take too much time and has often negative consequences on students' motivations; too less fundamentals will limit the students' abilities to apply the formal methods to problems not equal but only similar to the ones where they have used it first. The goal is to prove that formal methods can be used successfully to solve real world problems, and to enable the students to dive into a subject if necessary.

At the end of the course each group has to present its solution to the problem as well as the principal points of the theoretic topic acquired.

3.4 Software Engineering

The software engineering course is held in a more traditional way. Its main emphasis is on analysis and design of systems (and not on the programming part). Normal lecture is complemented by exercises where the students in groups have to do the requirements analysis of a not too small software project. Here the size of the problem is essential: Nearly all of the methods and techniques used in software engineering only make sense when applied to sufficiently large projects. Small toy projects will never convince students of the necessity to apply analysis and design methods in a strict engineering manner. So the projects used for analysis are real world problems taken from professional work. Typical projects include a hotel reservation system, administration of a doctor's office, software system for a bus company, organization of a symphonic orchestra, traffic control for river navigation or a system for gross sales of special products.

For parts of this project a design has to be accomplished, too. We use a commercial UML tool for developing the requirements and the design document. The possibility of incorporating more or less formal definitions of requirements and constraints is explained. In the setting of the course this can be done by writing OCL expressions in the slots provided by the UML tool. Normally it's hard to convince students of the usefulness of such early formalization - but this is the moment where one has to be persuasive, especially as this could create a motivating ground for the course on model checking.

Generally we must face a problem here: Software engineering is essentially a discipline that teaches methods for solving a formalization problem: From vague problem descriptions to programs written in completely formal programming languages. These methods usually lead the software developer in a step by step mode from the informal to the formal, because doing the formalization is a very hard task and doing it in one big leap is impossible or at least very error prone. Usually requirements are defined in an early stage of the development process using natural language sentences and are therefore not very formal. Only later on through functional specification, design and coding more and more formal versions of the system are produced.

We must emphasize here that formalizing requirements at an early stage of the development process has different reasons: We want to provide a formal basis independent of the real programming process that will enable us to prove automatically that the requirements stated early in the process are fully met by the implementation. The more independent this formalization of requirements is from the actual programming process the better.

For practical reasons these formalizations can only be done for small parts of the project as full size practical problems are much too large to be done within the strict time limits of a university course. So we choose small parts of the overall system and have the students design the class structure. And for some of the methods of these classes preconditions, postconditions and invariants must be expressed in OCL. But as small toy problems can hardly show the usefulness of the approach we are still in search of appropriate examples. In our case of a study program in Information Security we will try to develop examples from fields where security and safety is an important part and hope to produce something useful for the upcoming semester.

Our experience so far with this course is that students appreciate the use of real world problems and usually understand that rigorous methods have to be applied to cope with the complexity of such problems. But it's much harder to get them doing a good job on the constraint definition in OCL.

3.5 Model Checking with TLA+

The introduction in Model Checking emphasizes first that in engineering usually systems are built upon some model. Later in the process it may be realized that the system implemented may be not quite what was envisioned right from the beginning of the undertaking and very often the system is revised but not reflected why the initial model was insufficient or failed. The result of this is that the cost of development is higher and the quality of the system is significantly lower.

Furthermore it is told that a much better way to support the development process is to start with a model with which one is able to check the important properties before the system is built and to change the model in case of failure and/or changes of system requirements at the model level. At this point the Batson (Intel) quote from 2002 (at the last cover page of [4]) which says that "TLA+ represents the only effective methodology I've seen for visualizing and quantifying algorithmic complexity in a way that is meaningful to engineers" is stressed, which then leads directly to dynamics of TLA+ [4] by demoing examples using the Toolbox [1].

The lecture continues by outlining some of the major system bugs caused by software failures to further enforce the motivation for the subject of Model Checking. After some state machine examples and their "translation" in TLA+ together with the checking features of the Toolbox we first try to emphasize the practical side of the Toolbox before giving some insight in the underlying mathematical concepts.

The experiences gained so far are:

– Over time the theory part became smaller and the focus successively went to the "discussion and demo" part based on the examples using the Toolbox.

- In particular, starting with simple examples right away (e.g. in the general introduction session at the first day!) has turned out to be an effective way to keep up students interest.
- These observations lead to the design change by adding a "discussion and demo" part at every block following the introduction.

Although a majority of students appreciate the shift from the lesser emphasize of the foundation towards the practical use of the Toolbox in demoing small examples, many have still difficulties to see the relevance for real world system development. Bridging the gap between examples useful for teaching and learning the basic concepts on the one hand, to there relevance for system development in general on the other hand seems to be a too big step for some students. This fact makes it even harder to get the necessary attention to the quite restricted minimal theoretical background on the mathematics seen to be required.

Recent feedback data from students highlighted the following items:

- Awareness of the limited mathematical background, but instead of proposing deepening the background more practical applications and more examples look to them as a better way to overcome this drawback.
- Examples presented are well understood, however the relevance to current industrial applications is lacking. Which companies are engaged in TLA+ projects? One asked, to what extend Microsoft is using TLA+ in their developments processes?
- One mentioned the lack of understanding, how TLC works "inside"? To what extent a deeper insight, a kind of look "under the hood" is important for understanding the Toolbox is currently under discussion.
- All emphasized a much stronger focus on practical examples leading to a better understanding to what extend TLA+ is or should be useful in and for industry.

Viewing the feedback as the current state so far (at least for universities of applied science!), the question is up, what has to be done next to reach a better understanding, by fully realizing the circumstances and drawbacks outlined?

Some of the above mentioned points may be covered by the following topics which are also most likely fruitful for teaching/learning in general:

- Extend the set of available examples by exchanging them within the TLA+ community, in order to widen basis for the "teaching by example" approach successively.
- Gain experience by applying the TLA+ Toolbox [1] to teach/learn the required mathematical foundations (e.g. by following and extending the approach already taken in the Hyper-Book!).
- Gain and evaluate feedbacks upon the self-learning experiences made by using the Hyper-Book (in conjunction with the Toolbox [1]).

The experience shortly tackled tells, that the Toolbox [1] is of great help in teaching and learning Model Checking with TLA+ [4], but should be used even more in providing the basics of the underlying mathematics in a very practical "hands on" fashion (e.g. by using a further version of Hyper-Book as a very useful step to foster self learning even more!). How to bridge the gap between the "worlds" of small examples to the real world is another issue for further study in general.

4 Summary

The key feature of teaching formal methods in universities of applied sciences is to raise an understanding of the necessity of theoretical concepts with the students. This can only be accomplished when theoretical and practical courses are tightly coordinated to give insight into the problems and the baselines of the techniques to their solution. Our approach tries to coordinate theoretical courses (Theoretical Foundations of Computer Science) and practical courses (Software Engineering) to provide a motivational basis for the course on Model Checking where we then combine theoretical knowledge with problems as close as possible to the future vocational practice of our students.

We are right in the implementation process of our course designs. We gained some experience within the last 2 years and we are now reworking the concept a bit and will put this reworked concept described here in execution this fall semester.

The ultimate goal of the approach is to convince students of the necessity of using formal methods and to enable them to apply these methods to real world problems when they go out into their profession. They are our ambassadors out there and the most important agents for enhancing the vocational practice. Or with the words of Kurt Lewin: "There is nothing so practical as a good theory" [6].

References

1. http://www.tlaplus.net/tools/tla-toolbox/. accessed: 2012-05-11.
2. Johann Haag Christiane Metzler. Effekte von Blockunterricht im Studiengang BSc IT Security. *Neue Wege gehen - Strategien und Modelle für Studien-, Lehr- und Lerninnovationen an der Fachhochschule St. Pölten.*
3. Michael Fisher Wiebe van der Hoek Clare Dixon, Mari-Carmen Fernández Gago. Temproral logics of knowledge and their applications in security. *Electronic Notes in Theoretical Computer Science*, 186:27–42, 2007.
4. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Pearson Education, Inc., 2003.
5. Sergey Bratus Michael E. Locasto Anna Shubina Len Sassaman, Meredith L. Patterson. Security applications of formal language theory. Technical report, 2001.
6. Kurt Lewin. Problems of Research in Social Psychology, in: Field Theory in Social Science. In D. Cartwright (Hrsg.), editor, *Selected Theoretical Papers*, page 169, 1951.

Contact the Computer Science Department for specif.Â College of Engineering & Applied Science. Menu. Main menu.Â Contact the Computer Science Department for specific degree requirements corresponding to a particular catalog year. 128 semester credit hours are required to complete this degree. Freshman Year. A Hypothetical Curriculum So, what are the requirements of our hypothetical computer science program?Â Our curriculum will attempt to integrate courses that would be common to all such programs, while also providing a selection of electives that could function as an introduction to those various concentrations.Â For example, all students who major in computer science at a given university may be required to take two general introductory courses in the field, but students who decide to concentrate on cryptography may be required to take more math classes, while students interested in electrical engineering may take required courses on robotics, while others interested in software development may be required to. * Students in Computer Science and Molecular Biology (VI-7) follow a variation of this curriculum scheme, adapted to accommodate the interdepartmental major. Useful links: Degree requirements for the NEW curriculum.Â You can access the Undergraduate and MEng degree programs checklist for the new curricula here! Course 6-1: Electrical Science and Engineering (Note: Old Curriculum). EECS undergraduate degree program 6-1: Electrical Science and Engineering. Course 6-2: Electrical Eng. & Computer Science. EECS Course 6-2: Electrical Engineering and Computer Science. Course 6-3: Computer Science and Engineering. Course 6-3: Computer Science and Engineering. 6-7: Computer Science and Molecular Biology. Curriculum and courses information for Computer Science.Â The Computer Science program has been designed to support and encourage participation in a variety of study-abroad programs. The majority of computer science majors who study abroad do so in one of four programs that have strong computer science offerings: The University of East Anglia in Norwich England, The University of Queensland in Brisbane Australia, The University of Otago in Dunedin New Zealand or with the Danish Institute for Study Abroad (DIS) in Copenhagen, Denmark.Â 364 Artificial Intelligence A survey of techniques for applying computers to tasks usually considered to require human intelligence.