Teaching Software Model Checking

Cyrille Artho[1]

Research Center for Information Security (RCIS)
National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan

Kenji Taguchi[2] Yasuyuki Tahara[3] Shinichi Honiden[4]

Information Systems Architecture Science Research Division
National Institute of Informatics
Tokyo, Japan

Yoshinori Tanabe[5]

Research Center for Verification and Semantics (CVS)
National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan

Abstract

The use of formal methods has become commonplace in hardware design, and is becoming increasingly widespread in software engineering. While formal methods have repeatedly been applied in safety-critical projects, their technologies and tools are not widely known, due to lack of in-depth education in current curricula. In this paper, we introduce the curriculum design of software model checking, which is part of a larger education program that addresses several issues in software engineering and formal methods in general. We will also touch upon the necessity of a formal methods body of knowledge (FMBOK) for the guidance of formal methods education.

Keywords: Formal methods, education, software engineering, model checking, body of knowledge.

1 Introduction

The use of formal methods is already widespread in hardware design and is becoming increasingly common in mission-critical software projects. Despite this success, software engineering courses typically only introduce formal methods at the Ph. D. level. In our education program called Top SE [7], we address issues in software

[1] E-mail: c.artho@aist.go.jp
[2] E-mail: ktaguchi@nii.ac.jp
[3] E-mail: tahara@nii.ac.jp
[4] E-mail: honiden@nii.ac.jp
[5] E-mail: tanabe.yoshinori@aist.go.jp

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

engineering and formal methods from requirements to testing, and provide a comprehensive education about these subjects to graduate students and engineers from industry.

Formal methods have been recognized as a rigorous software development methodology for safety-critical systems [13]. It is now recommended to be used in safety and security areas such as functional safety (IEC 61508) [8] and security assurance (ISO/IEC 15408) [9]. On the other hand, the teaching of formal methods is still at an early stage in the sense that there is no standardized curriculum guidance based on a body of knowledge on the whole area of formal methods.

In this paper, we introduce the curriculum design of our course in software model checking. In this course named "Model Checking of Concurrent Java Programs", cutting-edge research results are integrated with foundational materials on software model checking.

A survey on the undergraduate curriculum on formal methods was carried out by Oliveira [12] as a part of FME-SoE (Formal Methods Europe, subgroup on Education). The paper shows a wide variety of formal methods courses in Europe, but does not present a well-structured body of knowledge on formal methods. Unfortunately, there has not been any follow-up activity from this group since then, even though the demand for education in formal methods is growing constantly.

Formal methods could be taught within a broader topic (e.g., software verification [13]) or as a separate and independent course [10]. As we will explain later, our education program has a strong emphasis on formal methods. We provide a wide variety of courses ranging from model checking to formal specifications. When we assessed our courses on formal methods, we realized that the existing body of knowledge (BOK) on software engineering and computers was not helpful due to its lack of depth and width in its description of formal methods. This led us to pursue an idea of a body of knowledge specifically designed for formal methods.

This paper is organized as follows: The next section presents an overview of our education program. Section 3 describes the course on software model checking, which includes coverage of very recent research results. Section 4 discusses lessons learned while teaching this course for two years. The need for a body of knowledge on formal methods is explained in Section 5, while related work is covered in Section 6. Section 7 concludes this paper.

2 The Top SE program

The Top SE program is a non-accredited course at the Masters level, operated in close collaboration between industry and academia at the National Institute of Informatics in Tokyo [7]. The whole curriculum is fully funded by the Japanese government and comprises five lecture series as shown in Table 1. All courses in the five-lecture series are electives except for two basic courses, which are compulsory.

The program has the following distinguishing features:

- Cutting-edge software engineering technologies are covered comprehensively. The program encompasses software design patterns, aspect-oriented development, and model checking.

Table 1
Curriculum of Top SE.

Series	Courses
Requirements Analysis	Requirements Analysis
	Security Requirements Analysis
System Architecture	Component-based Development
	Software Patterns
	Aspect-Oriented Development
Formal Specifications	Formal Specifications (Foundations)
	Formal Specifications (Applications)
	Formal Specifications (Security)
Model Checking	Verification of Design Models (Foundations)
	Verification of Design Models (Applications)
	Model Checking of Concurrent Java Programs
	Verification of Performance Models
	Modelling and Verifying Concurrent Systems
Implementation	Testing
	Program Analysis
Basics	Basics of Software Science
	Practical Software Engineering

- Real case studies from industry are included. We regard our education program as a mission to transfer technology from academia to industry, and we tailor the program to meet the needs of industrial partners such as Hitachi, NEC, Toshiba, NTT Data, Fujitsu, and Nihon Unisys.
- The program emphasizes engineering practice and tools over theory and processes. Management and process aspects are not the focus of the program for the moment. [6]
- Collaboration with academic partners (Shinshu University, Tsukuba University, and others) ensures that our courses cover a broad spectrum. Multiple tools and methodologies are presented to teach not only the usage of tools, but also the techniques and guidelines that apply to practical software development.

As Table 1 shows, the usage of formal methods, including formal specification languages, in the early phases of the software development life cycle (requirements and design) is emphasized. Testing and program analysis are covered as well, with *Model Checking of Concurrent Java Programs* being the most advanced course in this area.

The compulsory course *Basics of Software Science* covers elementary logics (propositional and predicate logics), temporal logics (LTL, CTL), concurrency theory, abstraction, automata, model checking algorithms, and their implementation techniques. Traditionally, these topics are taught in different courses (e. g., preliminary mathematics, operating systems, and more advanced courses in theoretical computer science). Our course includes all necessary preliminary knowledge on formal methods, which makes it rather unique.

[6] A new plan is underway to include process/quality management courses.

Table 2
Module structure of the "Model Checking of Concurrent Java Programs" course.

Description		Lectures
JPF Basics	Introduction	1st
	Safety	2nd
	Liveness	3rd, 4th
	Fairness	5th, 6th
	Group Exercise	7th
Model Checking Networked Programs	Introduction, stub-based approach	8th
	Input/output caching, centralization	9th
	Centralization	10th
	Network layer for centralization	11th
	Group Exercise	12th

3 Course and curriculum design

In this section, we explain the course description shown in Table 2 in detail.

3.1 Goals of the course

The distinguishing features of model checking compared with other approaches such as testing or theorem proving are that it can be fully automated and that all possible behaviours can be checked [13]. Model checking technology can be applied to a design specification or to source code [6]. The aim of design analysis is to validate correctness of an early design. This aspect of model checking is taught in *Verification of Design Models (Foundations, Applications)*, as shown in Table 1. The direct analysis of source code, which is taught in this course, is a relatively new idea.

Verification of the final code is necessary if one wants to have full confidence in the final product. At the moment, we have to admit that heavy-weight tools such as software model checkers cannot handle large implementations yet. Still, when forsaking complete coverage of the state space (when looking for bugs rather than trying to show correctness), the Java PathFinder model checker (JPF) still manages to find complex failures in production systems that cannot be found by testing. The goal of the course is to make students proficient in this novel verification technology, and give them an understanding of its capabilities and limitations.

There are still only a few industrial practices where program code is directly model checked. Model checking the implementation directly has the potential to eliminate the process of creating an abstract model from source code. Nevertheless, due to the state space explosion problem, code usually has to be simplified (abstracted) prior to verification. In practice, verification has to focus on key aspects such as inter-thread communication and synchronization. Therefore, early prototypes (whose fine-grained design is not fixed yet) are ideal for model checking.

Our lecture does not focus on the software development process and therefore does not favor verification at a particular stage in the development cycle.

3.2 Model analysis

The first half of the course (seven lectures) is devoted to the basics of JPF. Most students have only learned the basics of model checking in classrooms and are not

familiar with software model checking in practice. Since they tackle verification of network programming in the second half of the course, they must master the basics of JPF in the first half. We also want the students to understand how the property patterns learned in the *Basics of Software Science* course, namely, safety, liveness, and fairness, are expressed and verified in the context of software model checking.

The first lecture is an introduction. A brief review of related topics learned in the *Basics of Software Science* course covers: the significance of verification techniques such as model checking, the usefulness of model checking in a concurrent or distributed environment, and advantages and drawbacks of software model checking. An overview of JPF is also presented.

Safety properties are the first target in the course. JPF is designed so that safety properties are verified in a natural way. In this part, we illustrate how they are verified with JPF by using simple examples.

The next topic regards liveness and properties in the form of Fp and $G(p \rightarrow Fq)$. Since they cannot be verified with the built-in functions of JPF, we implement a search extension for such properties. Through exercises, students learn that JPF is designed so that functions can be extended by preparing modules and how such modules are developed.

Finally, we consider fairness, which can be expressed in linear temporal logics (LTL). In the *Basics of Software Science* course, the students learned how LTL formulae are converted into Büchi automata [6]. In this course, we implement a search algorithm for JPF based on the conversion. Students study an application of automata theory through verification exercises on liveness properties under fairness.

In 2006, this part also included lectures on abstraction techniques such as data abstraction, predicate abstraction, and program slicing. However, these lectures were moved to the *Basics of Software Science* course for 2007.

3.3 Networked software

The second half of the course deals with networked software. At the time of course creation, state of the practice consisted of treating network functionality as an open call returning a non-deterministic result. This results takes all possible behaviours into account but may also generate spurious behaviours, which are not possible in practice. A less abstract stub function can be used as an alternative to a completely non-deterministic result. Such a stub returns a result that approximates the behaviour of the environment for a given test case. After briefly touching on automated, domain-specific techniques [5], the course teaches manual stub creation.

After coverage of stubs, the concept of a novel input/output caching approach is introduced. Because the implementation was in a prototype stage [4] when the course was taught, the remainder of the second half focuses on the centralization approach [15] and recent extensions that made it applicable to software communicating using TCP/IP sockets [2]. Besides the usage of JPF for analyzing the application, the theory and engineering behind the method are covered in great detail. The aim is to provide enough knowledge to allow the students to understand the tool in detail, and to cover the concepts of other communication mechanisms (such as pipes).

The exercises started with two simple server applications, which served as examples for stub usage and centralization. After that, a chat server with a main/worker thread architecture was used as a running example for the final exercises. This final exercise was rather complex, as the architecture of the chat server resembles the design of real-world servers. Hence, it constitutes a good preparation for analyzing real software using a model checker.

3.4 Model checking tools

The Java PathFinder model checker [16] (JPF) is the main tool for this lecture. We chose JPF because of its free availability, easy portability (Sun's Java Development Toolkit version 1.4 being the only prerequisite) and its facilities that allow extensions to be plugged in relatively easily. Furthermore, JPF version 3 is relatively mature and has no defects that would prevent us from carrying out exercises and case studies.

At the time when the lecture was made, no Java model checker could deal with network communication. This severely restricted the usefulness of JPF, as interesting Java programs typically require networking. During 2005 and early 2006, the necessary extensions and an additional tool, called *Centralizer*, were built to allow verification of networked software [2]. These research results were included in the lecture, because they greatly improve the applicability of software model checking.

4 Student outcome

In this section, we report on the student responses from three sources: coursework reports, students' presentations, and questionnaire results.

Coursework Reports

The students were asked to submit five coursework reports during the course. Some exercises were obligatory, and the others were non-obligatory. The former could be completed if they understood the basics of the lectures, while deeper understanding and skills were needed to complete the latter.

Almost all students submitted reports. About half of them submitted only the obligatory exercises, while a couple of students tackled almost all the exercises. Since 87 % of the obligatory exercises were answered correctly, we concluded that the students had acquired the basic knowledge. About a quarter of the students gave good answers for more than half of the non-obligatory exercises. We considered that these students had mastered sufficient skills to use JPF.

Students' Presentations

The seventh and twelfth lectures were group exercises. Students were divided into several groups, each of which consisted of five to seven students. Different tasks were assigned to each group. A group discussion of 60 minutes was followed by a presentation of 15 minutes per group.

One of the tasks in the first exercise regarded safety properties; the other was about liveness properties. The former was a classical "crossing a river" problem. Verification itself was simple. However, students were also asked to modify a VM listener so that additional information was added to the error trace. The next task concerning liveness was a variant of the dining philosopher problem. The exercise consisted of model checking, finding the reason for the breach of the liveness property, and modifying the program.

Since the task list was presented beforehand, motivated students had already started investigating the tasks, and the discussion was conducted under their leadership. Although no group completed all the assigned tasks, all groups found solutions to the exercises marked as obligatory.

During preparation of the presentation, it was observed that some of the students who had submitted only obligatory exercises in the coursework reports were not proficient in the use of the tool. However, even such students learned from other students through the discussion, since they actively took part in it. In the questionnaire described below, six out of seven students answered that the group exercise deepened their understanding.

The final exercise, discussed in the last lecture, was the most difficult one. It required both proficiency with JPF as well as an in-depth understanding of concurrency in Java. The exercise was presented one week prior to the group discussions. Some of the students who studied the material were able to complete this exercise, while others recognized some important points but did not arrive at a complete solution on their own.

Questionnaire Results

Students were asked to fill in questionnaire forms at the halfway point and at the end of the course; eight and seven students responded, respectively.

From their answers, we presume that the students were satisfied with the course – on a scale of 1 to 5, the average score for the lectures was 4.0, and the difficulty was judged to be 3.7 (1: easy to 5: difficult). The self-evaluation of proficiency score was 3.4, which was not much different from that evaluated by the lecturers.

The applicability of JPF to development work in their company was not evaluated very affirmatively — the score was 3.0. Among the free responses were: "JPF requires the user to possess a high-level knowledge," "automatic re-modeling of distributed environment is desired for easy usage," and "development of patterns is needed to apply the tool to actual problems." These comments reflect the current status of the software model checking technology. However, it is our opinion that development of the skills to utilize such an emerging technique is an advantage of "Top SE".

5 Outlook: Body of knowledge on formal methods

As shown in Table 1, formal methods are at the core of our education program. Therefore, we are very concerned with how to design the whole formal methods courses in a well-structured and organized way to ensure that all necessary knowledge areas are covered and all teaching materials are properly balanced.

To achieve this goal, we have come to realize that there is a lack of identification and specification of the underlying content of formal methods. These activities are definitely necessary for providing curriculum guidance for various formal methods courses. Indeed, formal methods appears to be a relatively mature discipline. Nonetheless, there is no established body of knowledge on formal methods. SWEBOK (software engineering body of knowledge) [1] is a comprehensive description of the software engineering knowledge standardized by IEEE CS and ACM, but it is too general for our purposes. A body of knowledge that is quite similar to our idea is PMBOK (project management body of knowledge) [14]. PMBOK was standardized by the Project Management Institute. It is widely used as the de facto standard reference for project management.

Our proposal here is to standardize a body of knowledge on formal methods (FMBOK) for general guidance of curriculum design on formal methods. We are at an early stage of developing this idea and we would like to call for an international effort to explore it. We believe this idea will be of a great benefit for all educators and practitioners working on formal methods.

6 Related work

There are a lot of courses treating model checking of programs. They can easily be found by WWW search engines such as Google, with keywords such as "software model checking", "course" and "lecture". However, most of them treat software model checking in general and are separated into small parts, each one of which is self-contained with an individual tool or technique. In addition, only a small number of the courses include exercises on practical problems. Kwiatkowska gave one of the exceptional example [11]. She assigned her students project work including case studies of verification activities. However, her work does not include any distributed system problems.

A shorter version of this course was also taught as a half-day tutorial at the Automated Software Engineering (ASE) conference in 2006 [3].

7 Conclusions

In this paper, we introduced the design and students' outcome of our software model checking course and touched upon the idea of a formal methods body of knowledge (FMBOK). Software model checking has not been widely taught yet. Several ways to teach it are possible. In order to access and compare courses, we will need a specific knowledge area for software model checking. Hence, our main future work is to develop FMBOK and its specific knowledge area for software model checking.

Acknowledgments

The Top SE program is fully supported by the Special Coordination Fund for Promoting Science and Technology, for fostering talent in emerging research fields by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] Abran, A. and Moore, J. W. and Bourque, P. and Dupuis, R.: Guide to the Software Engineering Body of Knowledge 2004 Version SWEBOK. IEEE (2004)

[2] Artho, C. and Garoche, P.-L.: Accurate centralization for applying model checking on networked applications. In Proc. 21st Int'l Conf. on Automated Software Engineering (ASE 2006), Tokyo, Japan (2006)

[3] Artho, C.: Model Checking Networked Software. Tutorial at ASE 2006, Tokyo, Japan.

[4] Artho, C. and Zweimüller, B. and Biere, A. and Shibayama, E. and Honiden, S. Efficient model checking of applications with input/output. Post-proceedings of Eurocast 2007, LNCS 4739:515–522, 2007. Springer

[5] Ball, T. and Podelski, A. and Rajamani, S.: Boolean and Cartesian Abstractions for Model Checking C Programs. In Proc. TACAS 2001, LNCS 2031, pp. 268–285, Genova, Italy, 2001. Springer

[6] Clarke, E. and Grumberg, O. and Peled, D. Model Checking. MIT Press, 1999.

[7] Honiden, S. and Tahara, Y. and Yoshioka, N. and Taguchi, K. and Washizaki, H.: Top SE: Educating Superarchitects Who Can Apply Software Engineering Tools to Practical Development in Japan. Proceedings of International Conference on Software Engineering (ICSE '07) 708-718, IEEE (2007)

[8] IEC 61508.: Functional safety of electrical/electronic/programmable electronic safety-related systems. Bureau Central de la Commission Electrotechnique International, Geneve, (2000)

[9] ISO/IEC 15408.: Information technology - Security techniques - Evaluation criteria for IT security - Part1, Part2 and Part3. ISO/IEC 2005 (2005)

[10] The Joint Task Force on Computing Curricula, IEEE CS and ACM.: Software Engineering 2004. IEEE CS and ACM (2004)

[11] Kwiatkowska, M.: Safety Critical Systems and Software Reliability.
`http://www.cs.bham.ac.uk/ mzk/courses/SafetyCrit/index.html`.
Lectures in University of Birmingham, School of Computer Science (2006)

[12] Oliveira, J. N.: A Survey of Formal Methods Courses in European Higher Education. CoLogNet/FME TFM'04, LNCS 3295 (2004) 235–248

[13] Peled, D.: Software Reliability Methods. Springer (2001)

[14] Project Management Institute.: The Guide to the Project Management Body of Knowledge (3rd edition). Project Management Institute (2004)

[15] Stoller, S. and Liu, Y. Transformations for model checking distributed Java programs. In Proc. SPIN 2001, LNCS 2057, pp. 192–199. Springer, 2001

[16] Visser, W. and Havelund, K. and Brat, G. and Park, S. and Lerda, F.: Model checking programs. Automated Software Engineering Journal, 10(2):203–232 (2003)