On Teaching Formal Methods: Behavior Models and Code Analysis*

Jan Kofroň[1,2], Pavel Parízek[1], and Ondřej Šerý[1]

[1] Charles University in Prague,
Department of Software Engineering Malostranské náměstí 25,
118 00 Prague 1, Czech Republic
{kofron,parizek,sery}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

[2] Academy of Sciences of the Czech Republic,
Institute of Computer Science Pod Vodárenskou věží 2,
182 07 Prague 8, Czech Republic
kofron@cs.cas.cz
http://www.cs.cas.cz

Abstract. Teaching formal methods is a challenging task for several reasons. First, both the state-of-the-art knowledge and the tools are rapidly evolving. Second, there are no comprehensive textbooks covering certain topics, especially code analysis. In this paper, we share our experience with teaching two courses. The first is focused on classics of modeling and verification of software and hardware systems (LTS, LTL, equivalences, etc.), while the other one involves topics related to automated analysis of program code. We hope that other lecturers can benefit from our experience to improve their courses.

1 Introduction

For a developer of a system with high demand on reliability (e.g., safety-critical systems and device drivers), at least a basic insight into formal methods is essential. In particular, familiarity with model checking and code verification techniques and tools is an important asset. First, it is useful when actually dealing with the tools, which often communicate in specialized formalisms (e.g., various temporal logics). Second, it helps developers to decide whether they can benefit from a concrete technique or tool and also to choose among different implementations to suit their specific needs. The latter point is especially important with the increasing number of available code analysis tools that are ready for industrial use, at least in specialized domains (e.g., Slam [8]). The underlying techniques have different strengths and limitations, which is very hard to assess without a deeper insight.

Well targeted formal methods education of the future software developers is very important, but also very intricate. In order to fully understand the topics, quite deep mathematical background (e.g., in logics, algebra, and automata

* This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838) and by the Czech Academy of Sciences project 1ET400300504.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 144–157, 2009.

theory) is required. This is in contrast to the current trend of a slow decrease in the amount of mathematical theory taught in favor of software development practice. As a result, building a formal methods course based on the limited foundations is very challenging.

Another obstacle is lack of literature. When it comes to general modeling of systems and model checking, *Model checking* by Clarke et al. [24] forms an excellent basis for a course. Unfortunately, the situation is not so good in the case of code analysis (as applied in tools like Java PathFinder [4] and Blast [2]). This topic is relatively new and still rapidly evolving. To the best of our knowledge, there is no comprehensive publication summarizing and comparing the different techniques so far. This means that a course has to be based mainly on various journal and conference publications and technical reports. These publications are rather brief, as the page range is typically limited. They also often differ in the level of abstraction and the underlying formalization and notation. Except for the additional preparation overhead, it is not an issue for the lecturers; however, such a form of study materials constitutes a major obstacle for the students.

In this paper, we share our experience with teaching formal methods in the scope of a new master study plan, *Dependable Systems* (Sect. 2). Among other courses, the plan contains two one-semester courses on formal methods that together cover modeling systems, model checking, code verification, deductive methods, and static analysis. The common goal of the courses (and the study plan in general) is balancing the theoretical and practical skills of the students. In Sections 3 and 4, the topics covered by the two courses are summarized in more detail along with the main references to study materials and the list of tools the students get acquainted with. We believe that other lecturers preparing similar courses will benefit from our experience. Sect. 5 contains observations and points to be discussed by the formal methods teaching community.

2 Our Vision and Realization

In 2008, a new study plan for master studies, *Dependable Systems*, emerged in our department as a reaction to both increasing industrial demand for highly specialized software experts and differentiation in their expected knowledge. The motivation of the new study plan is to provide industry with graduates familiar with techniques necessary to develop dependable systems (e.g., embedded, safety-critical, and real-time systems).

On one hand, this includes rather low-level knowledge of system architectures, operating systems, middleware, real-time systems, embedded systems, and parallel computing. On the other hand, the graduates get insight into software architectures, component systems, and services. Of course, courses on formal methods are a natural part of this study plan as well.

In general, the Dependable Systems study plan provides students with both the theoretical foundations and the practical hands-on experience with different tools and development techniques for more exotic platforms (e.g., embedded devices) and development under specific conditions (e.g., real-time, limited memory).

Considering formal methods, there are two specialized lectures: *Behavior Models and Verification* (Sect. 3) and *Program Analysis and Code Verification* (Sect. 4). The former covers modeling and model checking of software and hardware systems. The latter specializes on techniques for direct code analysis. Originally, there was only the former course covering also few code model checking topics. However, the amount of information associated with recent advances in code model checking and success of projects like Slam[8] in practice simply did not fit into a single course and motivated us to separate the course into the two specialized courses.

3 Behavior Models and Verification (NSWI101)

The course *Behavior Models and Verification* [14] aims at providing basics of the behavior modeling of systems and their consequent verification. The attendees of the course, future developers, should learn about the principles of formal specification and verification as well as work with state-of-the-art tools that are used in industry for verification of hardware and software models nowadays. Since this is an introductory course for master's level students, we do not assume any special knowledge in this area in addition to what they have learned during their bachelor's level studies. In particular, this includes propositional and predicate logics, and the automata theory. After passing the course, the students should be able to construct a formal behavior specification of a simple hardware/software system, think of and specify properties of interest, and, eventually, verify these properties using available tools. As to the organization, there are a lecture and a lab every week.

3.1 Lectures

In Fig. 1, the topics covered by the lectures are depicted; the main body includes the following:

- **Basic concepts.** LTS, Kripke structure, and different preorder and equivalence relations
- **Temporal logics.** Syntax, semantics, and expressive power of LTL, CTL, and CTL*
- **Model checking algorithms.** Both explicit (for LTL and CTL) and symbolic (for CTL) based on OBBDs
- **Partial order reduction.** The Ample set algorithm

This part of the course is motivated mainly by the comprehensive book *Model checking* by Clarke et al. [24]. A suitable level of abstraction is maintained throughout the book, which makes it also a useful study material for the attendees.

The rest of the lectures introduce timed automata and process algebras. The overview of timed automata is motivated by [16] and presents the basic properties

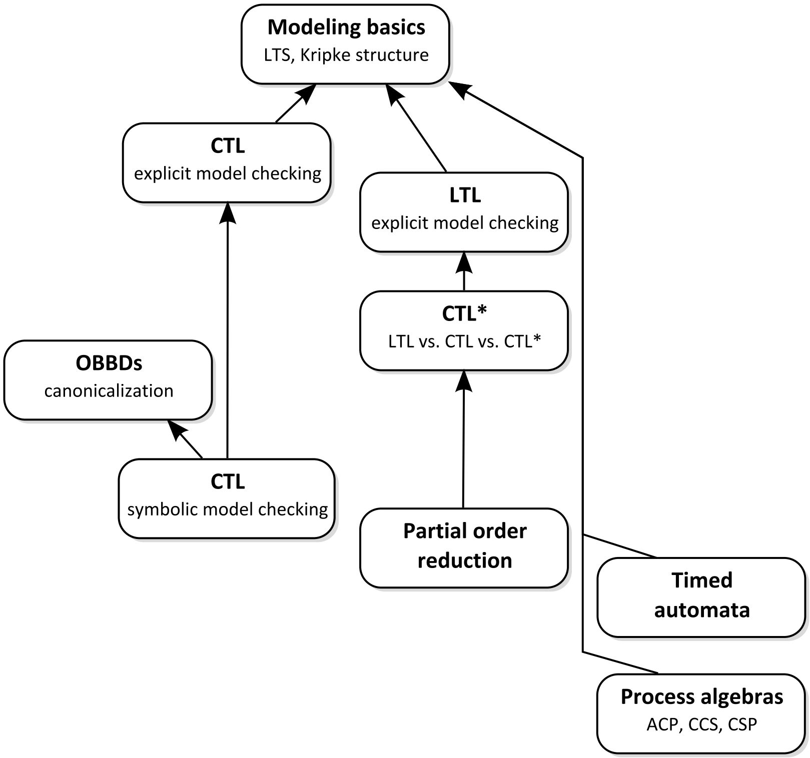

Fig. 1. Topics covered by the NSWI101 course and their dependencies. The corresponding lectures are held in the top-down order.

of the class of timed regular languages, emptiness check algorithm, parallel composition, and references the Uppaal integrated environment [12]. The lectures on process algebras focuses on *Algebra of Communicating Processes* (ACP) [21] and its content is highly inspired by the book *Introduction to Process Algebra* by Fokkink [26]. As an example of a relatively recent application, the formalism for behavior specification of software components, *Behavior Protocols*, is also presented based on [15].

3.2 Lab

There are two major aims of the lab of the NSWI101 course—first, the students should practically exercise the algorithms and techniques presented during the lectures, and, second, the model checking tools are presented and their input languages are discussed in detail.

The first three labs are devoted to the Spin model checker [11] and its input language Promela. We use the slides from the official Spin website [38,39], which turned out to be very good for first understanding of the basic modeling concepts. The presentation of the language is divided into two parts. After each part, the students solve simple assignments during the lab, such as the modeling of the producer–consumer problem. The goal of these labs is to cover almost the entire language. The principles of model simulation and verification are also presented as well as the majority of the options (command-line switches), however, the details on implementation of the algorithms inside Spin are mentioned just briefly or entirely skipped. The tool is demonstrated using both command-line and graphical user interfaces, whose options and settings are briefly explained. For more information, the students are pointed to the complete slide sets and the Holzmann book on Spin [30].

The subsequent three labs are devoted to exercises of the techniques and algorithms presented in the lectures. This includes in particular modeling simple systems via LTS, deciding on equivalences of temporal logics formulae, representing formulae, sets, and Kripke structures via OBDDs, and LTL and CTL model checking algorithms. After the lectures introducing OBDDs and algorithms for symbolic CTL model checking, a lab focusing on SMV model checker, in particular NuSMV [5], is held. The tool along with the parallel assignment language is introduced and the students again try to model simple systems (e.g., dining philosophers and the producer-consumer problem) to become familiar with the tool.

The rest of the labs is again devoted to exercises related to the theory presented in the lectures. To provide an opportunity to work on homework assignments, there are two to three labs left out at the end of the semester.

There are two graded homeworks; the grading forms 55% of the final grade, while the rest, i.e., 45%, is formed by the grade a student gets from the final test. The first homework is assigned at the end of the fourth lab. The assignment is articulated in a very general way, such as "model a railway station" and "model an airport". This way turned out to be beneficial from several points of view. First, it is easy to reveal a potential disallowed collaboration of the students—such a general assignment is very unlikely to be "implemented" similarly in multiple cases. Second, the students are forced to think of suitable abstractions to be used to create the models. Third, they have to think of properties to check—this is very important according to us, since in most papers and text books, usually only deadlocks and in better cases also response patterns are considered. We believe, however, that it is important to verify specific properties that can be a matter of interest in particular cases. Nevertheless, the students are provided with examples of entities that can be modeled, the properties that can be checked, and ways to make their models simpler if they reach the limits of Spin, usually in the sense of the size of their state spaces. The maximum amount of points a student can normally get for the first homework is 40. However, if a student creates an exceptional model, he or she can get up to 5 extra points.

At the end of the seventh lab (slightly after the middle of the semester), the second homework is assigned. It is aimed at practicing the NuSMV tool [5]. Since the parallel assignment language is rather low level in comparison with Promela, we decided for a simpler assignment, in particular, modeling and verification of properties of a well-known algorithm, e.g., the Dekker's algorithm for mutual exclusion [25]. Because of the lower complexity of the second assignment, the maximum amount of points in the case of the second homework is 15.

3.3 Grading

The grade for the course is based on points. We award 0–40 and 0–15 points for the first and second homework, respectively, and 0–45 points for the written exam; the total number of points is therefore 100. The grading scale is defined as follows:

- Score of 80–100 points corresponds to the *excellent* grade.
- Score of 71–79 points corresponds to the *very good* grade.
- Score of 62–70 points corresponds to the *good* grade.
- Score of 0–61 points corresponds to the failure, i.e., to an unsuccessful attempt to complete the course.

The grading scale is defined to force students to do both homeworks and the written exam that is devoted to theoretical background, principles, and important algorithms; it is not possible to do just the homeworks or the exam to complete the course.

There are soft deadlines for both homeworks—after a deadline passes, for each day of delay, there is a penalty of 10% of the points awarded.

3.4 Experience

Originally, there was a lecture every week, while the lab was held every other week only, i.e., there were six labs during the semester. They were entirely focused on the tools and their input languages (Spin, NuSMV, Bandera). After two years, we realized that the students are quite able to construct models in the sense of using a specification/modeling language and corresponding tools to verify their properties, however, they did not capture the algorithms and underlying theories well. This is mainly due to the rather demanding amount of theory. Therefore, we decided to extend the course and have the lab every week to practice it.

The aforementioned fact that the students were able to create models and use the tools deserves more explanation here. Since for most of the students, this was the very first experience with behavior modeling, which differs from ordinary implementation, indeed, several problems occurred. According to the complexity of the models they submitted as solutions to the homework assignment (taking just the first Promela assignment into account), the students could be divided into two groups. The students of the first group submitted a sort of simplistic models which can be verified by Spin in a reasonable amount of time (in the order of minutes on a decent machine), while the others ran into difficulties with verification due to complexity of their models. The unfortunate conclusion of some of them was then that Promela is not a suitable modeling language, and that behavior modeling in general does not make much sense. After getting this kind of feedback, we have started to emphasize the goals and success stories of modeling in general and focused more on guidelines on how the models should be constructed, especially choosing an appropriate level of abstraction.

As to the organization of the course, after first two years, we have decided to teach this course in English, which is not a native language of the students. There were several motivations for doing so. First, the majority of the terms have just an English form and there are no widely accepted translations. Second, since the English-language skills are generally not on a proper level in our country, the students can benefit from knowledge of the English language at conferences and workshops in the future. Even though there was a significant drop in the number of students attending the course after switching to English, all the students attending the courses so far have given us a positive feedback on this issue.

4 Program Analysis and Code Verification (NSWI132)

While the course *Behavior Models and Verification* (Sect. 3) focuses on general behavior modeling of software and hardware systems and checking of various properties of the models, in the course *Program Analysis and Code Verification* [34] we focus on analysis and verification of programs in mainstream languages like Java and C. The goals of the course are twofold:

1. to show the students, future software developers, that there exist tools for formal verification and analysis of programs that can discover real bugs (errors) in non-trivial programs and/or verify many interesting properties in the programs, and to let students gain experience with usage of the tools;
2. to provide the students with basic knowledge of key approaches to program analysis and verification, and of advantages, challenges, and limitations associated with each approach.

Our vision is that students attending the course should be able to use the appropriate methods and tools during the software development process.

We do not expect any specific prior knowledge from the students—we only assume that all students have basic knowledge of the automata theory and predicate logic. Since the course aims at master's level students, and courses on the automata theory and logic are taught at the bachelor's level, all students should have the required knowledge. Moreover, we do not strictly require that students complete the course on *Behavior Models and Verification* before participating in this course.

As for organization, the course run in the winter of 2008 for the first time. There was a lecture every week and a lab every other week. We give the students three homework assignments as a part of the course.

4.1 Lectures

The goal of the lectures is to introduce and describe the main approaches to verification and analysis of programs (code). We divided all the lectures (Fig. 2)

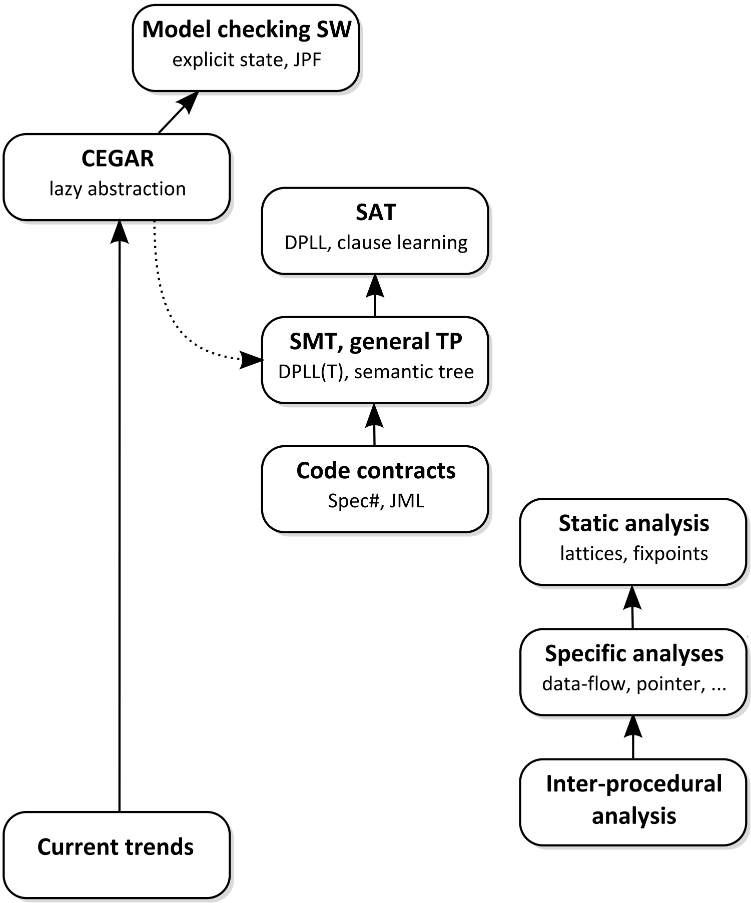

Fig. 2. Topics covered by the NSWI132 course and their dependencies. The corresponding lectures are held in the top-down order.

into four blocks: *Program Model Checking*, *Deductive Methods*, *Static Analysis*, and *Current Trends*. In the lectures forming each block, we describe theoretical background (e.g., lattices and fixed points for static analysis) of the approach, the basic principles and concepts of the approach (e.g., state space traversal in case of program model checking), and main limitations and challenges associated with use of the approach (e.g., state explosion) together with some solutions to the challenges (e.g., POR and symmetries for state explosion).

Program Model Checking comprises lectures on both explicit-state program model checking and CEGAR-based algorithms. The explicite-state program model checkers are explained on the example of Java PathFinder [4].

The lectures are based on a related paper [41] describing the algorithms Java PathFinder uses for POR, efficient state caching, etc.

The CEGAR-based algorithms are explained on the examples of Slam [8], SatAbs [7], and Blast [2]. The lectures cover predicate abstraction, abstraction refinement loop, and lazy abstraction; they are based on the papers [18,22,37] and nice tutorial slides on lazy abstraction [29]. Note that, before diving into the theorem proving details in the next lectures, the theorem prover is used here as a black-box with emphasis on the type of questions it is able to answer.

Deductive Methods and their application in program verification. In this block, we provide an overview of the main techniques used in SAT solvers, SMT solvers, and general theorem provers. The lectures are inspired mainly by the books *Decision Procedures: An Algorithmic Point of View* by Kroening and Strichman [31], and *Automated Theorem Proving: Theory and Practice* by Newborn [32]. Additional information on SAT solvers was taken from a nice survey [42].

In the subsequent lectures, we present application of solvers in the contract-verification frameworks like ESC/Java2 for JML [3] and Boogie for Spec# programs [10]. Here, the most useful information sources are the papers that describe algorithms used in Boogie [20,19].

Static Analysis block contains methods that are based on the lattice theory and computation of fixed points. This includes traditional data-flow analyses, points-to and shape analysis, and also a brief introduction to a control-flow analysis. The structure and content of this block of lectures is to a great extent based on the lecture notes [40]. A more thorough source is the book *Principles of Program Analysis* by Nielson et al. [33], however, we found it a bit too formal for use in an overview course.

Current Trends in formal analysis and verification of programs are summarized in the last block. Here, we provide an overview of the very recent topics based on various conference papers. The topics include compositional verification using assume-guarantee reasoning [27], symbolic execution in Java PathFinder [36,17], and a combination of testing with predicate abstraction [35,28,23].

4.2 Lab

The main goal of the labs is to provide the students with a hands-on experience with selected tools for verification and analysis of programs (code).

Each lab is devoted to a specific tool—for example, Java PathFinder and Soot framework [9]. First we explain how a tool works, how it can be configured and executed, and how to prepare its input and interpret its output—all this on simple demo programs and examples. Then, in the second part of a lab, we assign some simple tasks to the students, so that they can get their own experience with using the tools (and "playing" with them).

We present a single tool for each main technique (method) that is described in the lectures. To be more specific, we present the following tools:

- Java PathFinder [4], an explicit-state model checker for Java programs,
- Blast model checker [2], an implementation of the CEGAR-based model checking algorithm,
- SatAbs [7], model checker for C programs that uses CEGAR and a SAT solver,
- PicoSAT [6], a state-of-the-art SAT solver,
- Yices [13], a state-of-the-art SMT solver,
- ESC/Java2 [3], a tool for verification of Java programs against JML specifications [1], and
- Soot [9], a framework for static analysis and transformation of Java programs.

We have selected these tools due to their maturity and stability, moreover they are widely used, and open source.

The homework assignments directly follow the labs. Our motivation behind the homeworks is to let students try the tools on a larger example (program) than it is possible during a lab. There are three homeworks together. The theme of the first homework is Java PathFinder—the students are required to create a reasonable abstract environment for an open system and also to create a custom property. In case of the second homework, students are required to create a JML specification for several Java classes and to verify the classes' implementation against the specification. Finally, the third homework consists of creating custom analysis and transformation of Java source code on top of the Soot framework. The time needed for each homework is between 8 and 16 hours, depending on student's skills.

4.3 Grading

The grade for the course is based on points. We award 0–10 points for each homework and 0–30 points for the oral exam; the total number of points is therefore 60. The grading scale is defined as follows:

- Score of 49–60 points corresponds to the *excellent* grade.
- Score of 40–48 points corresponds to the *very good* grade.
- Score of 31–39 points corresponds to the *good* grade.
- Score of 0–30 points corresponds to the failure, i.e., to an unsuccessful attempt to complete the course.

We have defined such a grading scale in order to force students to do both homeworks, which are about practical use of the tools, and oral exams that is devoted to theoretical background, basic principles of the approaches and important algorithms. In particular, it is not possible to do solely the homeworks or solely the exam to complete the course.

4.4 Experience

After the first year, our experience with the course is somewhat mixed. On the one hand, the students were interested in the discussed topics and we were very satisfied with the quality of students' solutions to the homework assignments.

On the other hand, we found that having a lab only once per two weeks is not enough, similarly to the other course. For the upcoming years, we plan to have a lab every week. Some of the labs will focus on manual computation of the key algorithms using paper and blackboard, while the others focus on practical experience with the tools.

5 Evaluation and Discussion

The common issue of both courses is a low number of students attending the courses. We believe that the low attendance (enrollment) of students has the following two main causes:

- The usefulness of formal methods in industrial software development is not obvious to the students. They probably do not see the benefit of formal verification and analysis in comparison to testing. Moreover, formal methods are rarely used in software companies and therefore the students are not forced to learn about them. The students prefer to attend those courses, which they see as useful for their employment (XML, software engineering, web development, etc.).
- Courses on formal methods typically require significant mathematical background (logics, automata theory, formal languages, etc.) and they are also typically more demanding than the courses on XML and web development. Since most students prefer to choose the easier way to get the degree, they tend to avoid mathematics as much as possible.

While we see these two causes as general, they may be specific to our university to a certain extent.

Another problem is the lack of literature about formal methods at the level of master's level studies. There are many books and research papers that could be used, however they are often aiming at PhD students and researchers. This is especially problematic in the course *Program Analysis and Code Verification*, since we are not aware of any comprehensive book on code analysis and verification—e.g., with an extent and coverage similar to [24], which we use in the course *Behavior Models and Verification*.

As for the structure and syllabus of the courses, as the greatest benefit for students, we see the possibility to get hands-on experience with the tools and see their advantages and limitations, since they will not have such an opportunity in industry. Nevertheless, they are not always able to assess the complexity of models (or programs) with respect to formal analysis and verification, even after completing the courses—this requires years of experience with the practical application of formal methods.

6 Conclusion

In this paper, we have shared our experience with teaching formal methods in the scope of a new study plan, *Dependable Systems*. We have presented the content of two courses—"Behavior Models and Verification" and "Program Analysis and Code Verification". While the former one is probably similar to courses at other universities regarding its structure and content, we believe that a reader will benefit from our experience with the latter one. We have structured the code analysis course in a way to provide the students with experience with more tools rather than introducing few tools in greater depth.

Interestingly enough, the number of students attending the two courses is rather low in comparison to practically-oriented software engineering courses. Having a positive feedback from our students on the content and quality of the courses, we believe that the low attendance is caused by the fact that most students are interested in different topics.

As for the future, we plan to improve the first course by making it more comprehensible for students via including of more examples during lectures. As to the code analysis course, we definitely plan to stay up-to-date and update the content according to result of recent research in the area.

References

1. Java modeling language (JML), http://www.eecs.ucf.edu/~leavens/JML/
2. Blast project, http://mtc.epfl.ch/software-tools/blast/
3. ESC/Java2, http://kind.ucd.ie/products/opensource/ESCJava2/
4. Java PathFinder, http://javapathfinder.sourceforge.net/
5. NuSMV, http://nusmv.irst.itc.it/
6. PicoSAT, http://fmv.jku.at/picosat/
7. SatAbs tool, http://www.verify.ethz.ch/satabs/
8. Slam project, http://research.microsoft.com/en-us/projects/slam/
9. Soot framework, http://www.sable.mcgill.ca/soot/
10. Spec#, http://research.microsoft.com/en-us/projects/specsharp/
11. Spin, http://spinroot.com/spin/whatispin.html
12. Uppaal integrated environment, http://www.uppaal.com/
13. Yices, http://yices.csl.sri.com/
14. Adámek, J., Kofroň, J., Plášil, F.: NSWI101: Behavior models and verification, http://dsrg.mff.cuni.cz/teaching/nswi101/
15. Adamek, J., Plasil, F.: Component composition errors and update atomicity: static analysis: Research articles. Journal of Software Maintenance and Evolution: Research and Practice 17(5), 363–377 (2005)
16. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
17. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
18. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev. 40(4), 73–85 (2006)

19. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
20. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2005, pp. 82–87. ACM, New York (2005)
21. Bergstra, J., Klop, J.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
22. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R., Beyer, D.: The software model checker blast: Applications to software engineering. Int. J. Softw. Tools Technol. Transfer, 505–525 (2007)
23. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp. 29–38. IEEE Computer Society Press, Los Alamitos (2008)
24. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
25. Dijkstra, E.W.: Cooperating sequential processes. In: Programming Languages: NATO Advanced Study Institute, pp. 43–112. Academic Press, London (1968)
26. Fokkink, W.: Introduction to Process Algebra. Springer-Verlag New York, Inc., Secaucus (2000)
27. Giannakopoulou, D., Pasareanu, C.S., Cobleigh, J.M.: Assume-guarantee verification of source code with design-level assumptions. In: 26th International Conference on Software Engineering (ICSE 2004), pp. 211–220. IEEE Computer Society, Los Alamitos (2004)
28. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT 2006/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 117–127. ACM, New York (2006)
29. Henzinger, T. A., Jhala, R., Majumdar, R.: SPIN Workshop 2005 – Blast tutorial slides,
`http://www.cs.ucla.edu/~rupak/Powerpoint/BlastTutorial/SPIN2005.ppt`
30. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
31. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
32. Newborn, M.: Automated Theorem Proving: Theory and Practice. Springer, Heidelberg (2001)
33. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
34. Parízek, P., Šerý, O.: NSWI132: Program analysis and code verification, `http://dsrg.mff.cuni.cz/~parizek/teaching/proganalysis/`
35. Pasareanu, C.S., Pelanek, R., Visser, W.: Predicate abstraction with under-approximation refinement. Logical Methods in Computer Science 3(1) (2007)
36. Pasareanu, C.S., Visser, W.: Verification of java programs using symbolic execution and invariant generation. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 164–181. Springer, Heidelberg (2004)

37. Ranjit, T.H., Henzinger, T.A., Jhala, R., Majumdar, R.: Lazy abstraction. In: POPL, pp. 58–70. ACM Press, New York (2002)
38. Ruys, T.C.: SPIN Workshop 2002 – SPIN beginners' tutorial, http://spinroot.com/spin/Doc/SpinTutorial.pdf
39. Ruys, T.C., Holzmann, G.J.: SPIN Workshop 2004 – advanced SPIN tutorial, http://spinroot.com/spin/Doc/Spin_tutorial_2004.pdf
40. Schwartzbach, M.: Lecture notes on static analysis, http://www.brics.dk/~mis/static.html
41. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)
42. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 17–36. Springer, Heidelberg (2002)