Foundations of Semantics and Model Checking in a Software Engineering Course

Henning Bordihn[1], Anna-Lena Lamprecht[1], and Tiziana Margaria[1,2]

[1] Institute of Computer Science, University of Potsdam, Germany
{henning,lamprecht}@cs.uni-potsdam.de
[2] University of Limerick and Lero - The Irish Software Research Centre, Ireland
tiziana.margaria@lero.ie

Abstract. Formal methods provide systematic and rigorous techniques for software development and we believe that they should be part of academic software engineering education. In this paper, we describe how we included a selection of formal methods in a foundational Bachelor-level Software Engineering course. We show how we introduce the basic elements of modeling and programming language semantics, and discuss how we address the theory and practice of model checking within the scope of the two semesters of the course.

1 Motivation and Organization of the Courses

As formal methods provide systematic and rigorous techniques for software development, we believe that their contribution is foundational to the formation of the next generation of software developers, and as such they should be integral part of academic software engineering education already at the Bachelor level. At the University of Potsdam, the courses Software Engineering 1 and 2 are mandatory for Bachelor students of Computer Science and related subjects in their second year, that is, after they have completed foundational courses on mathematics, algorithms, data structures and programming during their first year. Hence, these courses are suitable to introduce basic formal methods education into the standard B.Sc. level curriculum. Starting in 2009, we have reformed the previously more traditional Software Engineering (SE) curriculum accordingly, which brought about a complete change of the philosophy and organization.

The outcome were the new SE 1 and SE 2 courses that we have refined and improved until today: aimed at teaching methods and mentality rather than facts and a collection of techniques, the new version made use of **agile**, **process-oriented techniques** from the very beginning, incorporating for the first time **formal methods** in the syllabus and integrating the **two terms of project experience** into the two-term course. In both terms, each week comprises 2h of lectures and 2h of tutorial/lab devoted to the discussion of problems and solving sample tasks during the session, without assignments given beforehand.

– In the first term, SE 1 (6 ECTS) focuses on design: the domain level and modeling languages, addressing and contrasting traditional vs. agile software development. It introduces model-driven (MDSD) approaches, domain-specific

languages (DSLs) as fundamental feature of MDSD approaches, and metamodeling as a tool for determining the abstract syntax of a DSL. While the focus is on the concepts and the modeling attitudes underlying these approaches and technologies, some of them are practiced using corresponding analysis and modeling tools.
- In the second term, SE 2 (6 ECTS) focuses on development: it emphasizes the technical level, with an introduction to IT project management, and software architecture, object-oriented design, testing, maintenance and reengineering. It addresses now in practice all the phases of software development described in the SE 1, supplemented by more modern topics like Design by Contract ("When the type system of the language is not enough"), Enterprise Application Architectures and Software Product Lines (SPL).
- In the concomitant project, students work together in small teams of 4-5 members to solve a more complex software engineering design (SE 1) and development (SE 2) task, applying to this case study the concepts and methods introduced in the lectures and the techniques and tools presented in the tutorials and labs.
- The course assessment consists of an individual final exam (written, 50% of the final grade) and the evaluation of the project deliverables: the final software product and the group reports (intermediate and final), in total again 50% of the final grade.

In this paper we present the course concept and report on our experiences from almost five years of conducting the course. In particular, we discuss how we address the theory and practice of formal methods on the basis of formal semantics and model checking. The paper is structured as follows: In the following section we explain in greater detail how we placed formal method contents in the reformed Software Engineering courses. Section 3 then focusses on the model checking aspects. Section 4 discusses the learning outcomes and competencies attained by the students of this course, and Section 5 concludes the paper with a review of experiences and some lessons learned.

2 Overview of the Formal Methods Contents

The new course aims at a foundational and experiential education, for which we included four formal methods blocks.

1. **Model Checking** [2, 7]: We start with the need to define "user stories" that concretize a system's behavioral specifications as process descriptions, for which we introduce process modeling languages. In order to convey the idea of property-based correctness and compliance, we directly introduce temporal logics and model checking as a formal verification technique on the (process-) model level, together with a corresponding tool. Regarding the models, the students build upon the notion of finite automata they already met in the first year in Theoretical Computer Science. Regarding the properties, they have prior knowledge of propositional logic.

2. **Formal Program Semantics**: As the process stories end up defining a collection of domain-specific actions and functionalities, the course then introduces DSLs and looks at the principles of formal semantics for programming languages. The definition of structural (small step) operational semantics (SOS) for a simple "While" language is treated as example. This language captures the control structures (excl. fork/join parallelism and hierarchy) the students used in their process models, and it corresponds to the flattened process models analyzed by the model checker. How to generate a control flow graph of a procedure from its stateless operational semantics is demonstrated in the lecture and practiced in the lab.
3. **Calculus of Communicating Systems and its SOS** (CCS) [6]: with the growing practical relevance of concurrent processes, we introduce CCS as an alternative formal model for distributed communicating processes. The definition of CCS and its SOS semantics (that now captures simple concurrency) is illustrated with several simple examples, then we introduce the algebraic laws of the calculus. We expand the catalogue of properties the students encountered so far with a discussion of several **equivalence notions** such as trace or equivalence for formal languages, bisimulation, and behavioral congruence. We discuss substitutability of (process or software) components based on indistinguishable behaviors.
4. **Static Type System**: in terms of properties, and compatibility checking of data structures and behaviors that operate on data, types and type checkers are the most successful technology. As illuminating example of how the success of software projects strongly depends on the choice of appropriate technologies, we discuss the role of the static type system for type-safe programming. We first show some sample effects that may occur when using script languages, and compare them to the behavior of compiled languages in similar scenarios. We use the **type determination/checking rules** for a simple expression language to illustrate benefits and limits of static type systems and prove simple type judgements using these type rules. Using the formalism, we show what is guaranteed about successful evaluations of expressions at runtime, and discuss which runtime errors may still be thrown.

3 Model Checking in Theory and Practice

Model checking [2, 7] provides a powerful property-based mechanism to analyze and verify static aspects of (arbitrary) behavioral models of a system. Generally speaking, it can be used to check whether a model M satisfies a property ϕ, usually written as $M \models \phi$, where ϕ is expressed in a modal or temporal logic.

We introduce it in the context of model-based software development, as an early detection technique that supports validation in the specification and design phase, prior to implementation. We show how it is useful to analyze global model properties, where syntax or type checking at the component level is not sufficient. Constraints are checked at modeling time, without execution, which offers another range of addressable issues in addition to local validation and

usual debugging methods. We also show that the list of properties against which the model is evaluated is easily extensible: including a new constraint in the verification only requires to provide a formula expressing the property of interest.

Consistent with our hands-on experiential approach, we introduce early in the course the jABC process modeling framework [8] as a technology that supports the development of executable process models as well as the verification of static model properties by using the GEAR model checking plugin [1].

3.1 Theory and Warm-up: Lecture and Lab Contents

Describing System Behavior The lecture introduces the basic concepts of process modeling, Service Logic Graphs and XMDD (eXtreme Model-Driven Development [5]), illustrating them with simple examples and one larger system (the travel authorization and refund processes at UP designed previously by other students). The students create the user stories as process models (called *Service Logic Graph*, or *SLG*) using process building blocks (called *Service-Independent Building Blocks*, or *SIBs*) from a service library in a drag&drop fashion, and connecting them with labeled branches representing the flow of control. As soon as the parameters of the SIBs have been configured, the process is ready for execution. Plugins add a wealth of additional features and capabilities to the jABC, ranging from simple execution by interpretation to more sophisticated functionality like code generation and workflow synthesis Our students use the SLG interpreter (the *Tracer*) for execution, and the GEAR plugin for model-wide evaluation of static properties (expressed in terms of modal or temporal specifications) directly on their own SLGs.

SLGs as Formal Models As suitable abstract model structure, we introduce Kripke Structures, Labeled Transition Systems and Kripke Transition Systems. Semantically, SLGs are in fact Kripke Transition Systems (cf. [7]) that combine classical Kripke Structures (cf., e.g., [2, Chapter 2]) with Labeled Transition Systems (cf., e.g., [4]) into model structures where both states and transitions are labeled. As such, they are directly amenable to formal analysis techniques, in particular to model checking.

Temporal Logic Properties as Constraints Starting from the propositional logic already familiar to the students, we introduce Hennessy Milner Logic (HML) [3], Propositional Linear Time Logic (PLTL, cf. [7]) and Computation Tree Logic (CTL, [2, Chapter 3]) as languages to express behavioral properties. While GEAR is internally a mu-calculus model checker, only CTL would be strictly necessary. We chose to include HML and PLTL primarily for didactic reasons, as they provide a good scaffolding for understanding CTL, especially regarding the role of the temporal operators along the paths and the path quantifiers. In the two associated labs, students practice the use of HML, PLTL, and CTL by modeling intuitive properties in terms of these logic languages and learning to interpret logical formulas in a comprehensible way.

Model Checking Finally, we explain how SLGs can be viewed as Kripke Transition Systems, how to use GEAR to assign atomic propositions to the SIBs of the SLGs, how to enter logical formulas, and how to check and debug

systems with GEAR. The use of GEAR is demonstrated by means of some simple examples, where the level of the student's independence is increased.

For these examples we use processes that are modeled as SLGs using only *Prototype SIBs*, like the abstract web shop process shown in Figure 1 (top). These SIBs are pure modeling means, as they do not have an actual implementation, but as the implementation is not relevant with regard to this kind of abstract model checking, they are sufficient for the purpose. Then system requirements are formulated as temporal properties. The lab discusses interactively which atomic properties (APs) need to be assigned to which nodes (SIB instances) and how to capture the formulated requirements as CTL formulas. The students use the *AP inspector* to annotate SIB instances with the atomic propositions, with results similar to the display in Figure 1 (bottom).

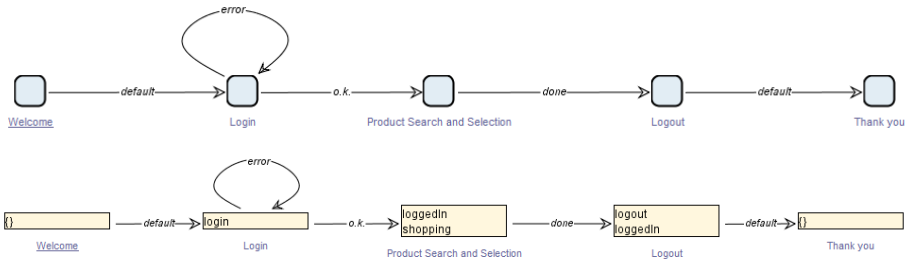

Fig. 1. The abstract web shop process (standard and Atomic Propositions view).

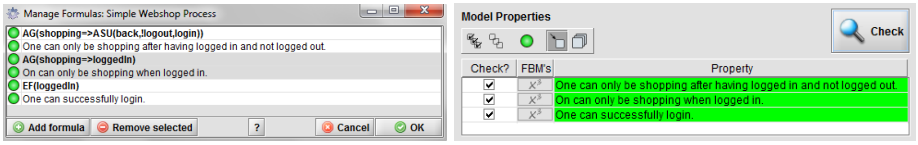

Fig. 2. Formula manager and model checking results.

With the GEAR formula manager, students can edit CTL formulas and document them with natural-language descriptions (see Figure 2, left). The model checking inspector displays all currently available properties, and one can toggle between the formulas and their high-level descriptions. In the hands-on sessions, students learn to check all properties, finding out which hold for the entire model (i.e. for the start SIB) as in Figure 2 (right), or individually for each node in the model. This check mode can also be applied to individual formulas: the nodes where the formula holds/does not hold are highlighted by green/red boxes respectively, as shown in Figure 3. Please note that the small examples in the

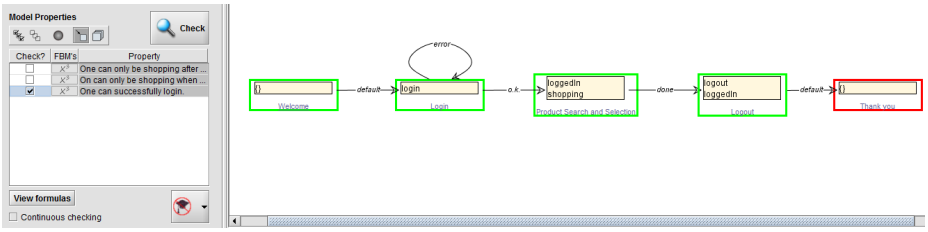

Fig. 3. Checking a single property for the whole model.

figures are meant for illustration purposes only, and that the examples on which the students work in their projects are larger and more complex.

3.2 Hands-on in Groups: Mastering the Project

After the lessons, tutorials, and introductory labs, the students are required to design a more complex jABC process, express relevant process properties, and check them with GEAR, in a project that mimics a system design from scratch. The concrete project task in SE 1 concerns first the user requirements for a sample Software Engineering project. Solutions are built in three steps:

1. Building a system specification via a UML Use Case Diagram supplemented by documented and prioritized use cases. The scenarios (system behaviors) corresponding to the use cases must be specified as process models, some as UML Activity Diagrams and some as SLGs. This way the students compare two exemplary modeling languages, learning to evaluate similar technologies wrt. adequacy for a specific purpose.
2. Developing a prototype of the system, delivering an executable SLG validated through the Tracer. For this process model, the students need to specify several global behavioral properties and verify them using the GEAR.
3. Documenting and evaluating the project activities and management decisions of the group, as a third part of their project solution and report.

In SE 2, the students continue the project towards the implemented product: structural design (architecture and class model) and a documented implementation, tested by unit tests. As it often happens in real software projects, in the second term they do not continue with their own specifications, but they must take over the specification from a different SE 1 team, whenever possible based on a different user story (i.e. a different system). They start over with the assessment of plausibility, feasibility, completeness, and quality of the previous group's deliverable and then they complete the specifications to a point where they feel comfortable with moving to the design and implementation.

4 Learning Outcomes and Observations

We describe the formal methods-related skills that the students acquire in our Software Engineering course in terms of the following competence fields:

– **Professional competencies**: Graduates of this course have broad knowledge about languages for the modeling of software. They know the principles and theoretical foundations of model checking.
– **Methodological competencies**: They are able to use selected languages and tools for the process- and object-oriented modeling of software. They can apply model checking for the formal verification of process models. They can assess/check the meaning of programs and their corresponding flow graphs using structural operational semantics (SOS). They are familiar with the use of logic calculi (axiomatic deduction systems).
– **Action competencies**: They are able to include formal verification methods in their software projects in addition to traditional validation and testing.

The course has so far been attended by over 250 students. Approximately half of the participants were enrolled in the B.Sc. Computer Science (2010-2014) or B.Sc. Computational Science (2015); the other half were students in the B.Sc. Business Informatics. There has been no observable difference between these groups regarding the achievements in the exercises, exams and projects. We saw however a significant difference among the skills developed by these students and those of the precedent years: While the previous generations were more confident in architectural issues and coding skills, our students have a more intuitive confidence with application development and with mastering a number of different technologies. We saw the advantage of this abstract and behavior-driven modeling in their approach to the subsequent module on Foundations of Service Engineering, where the black-box character of components is really central. The familiarity with the basic formal methods concepts was also advantageous in the subsequent specialistic course on Formal Methods in System Design (FMSD): taught in an e-learning fashion via teleconference-based lectures shared with the TU Dortmund and complemented by local tutorials and labs, it was clear that the Potsdam students had a smoother approach to the more technical material in FMSD due to their previous experience with the basic concepts.

5 Conclusion

We strongly believe that today, facing service orientation and an increasing interest in user-defined business processes, *lightweight formal methods* should be integral part of the standard academic software engineering education. Accordingly, since 2010 we have been teaching a foundational Bachelor-level Software Engineering course with four embedded formal methods components at the University of Potsdam.

We chose on purpose only so-called "lightweight" formal methods, i.e. such that algorithms exist (like type checkers and model checkers), and not theorem provers or proof techniques for program correctness like Hoare style program verification (both presented at length in other courses), because we wished to provide scalable approaches that have a low threshold to adoption, ease of reasonably confident use in the small, and a high ease of embedding in a full fledged model driven IDE (for us the jABC). While high-end Formal Methods continue

to require a deep knowledge of the mathematical concepts and formalisms, we believe that IDE-embedded FMs should be made available systematically to the next generation of Software Engineers as part of their basic profession-oriented education. Like basic English has successfully become within a generation a standard in the schools, establishing itself culturally as the "lingua franca" of international communication, the ability to take a domain-specific perspective, to think in terms of domain specific properties, to formulate them in such a way that both a user and a tool are able to work with them, and the ability to conduct verifications on early models of a system are in our opinion the key to a better understanding between IT professionals, in particular Software Engineers.

Another advantage is the ingrained sense of accountability and ownership that comes with the ability and habit of verifying early and verifying often, by means of tools that take an objective perspective (other than user inspections) and provide a third party repeatable outcome. With all the limitations and drawbacks that we know of the FM tools and approaches we have today, such a step towards accountability for decisions and outcomes early in the design constitutes in our opinion a central shift towards maturity of the professional figure, competence beyond the production of code, and ultimately towards establishing a new concept of *ethics and responsibility in the Software Engineering profession.*

References

1. M. Bakera, T. Margaria, C. Renner, and B. Steffen. Tool-supported enhancement of diagnosis in model-driven verification. *Innovations in Systems and Software Engineering*, 5:211–228, 2009.
2. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.
3. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, January 1985.
4. J.-P. Katoen. Labelled Transition Systems. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 615–616. Springer Berlin / Heidelberg, 2005.
5. T. Margaria and B. Steffen. Service-Orientation: Conquering Complexity with XMDD. In M. Hinchey and L. Coyle, editors, *Conquering Complexity*, pages 217–236. Springer London, 2012.
6. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1980.
7. M. Müller-Olm, D. Schmidt, and B. Steffen. Model-Checking - A Tutorial Introduction. In *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*, pages 330–354, 1999.
8. B. Steffen, T. Margaria, R. Nagel, et al. Model-Driven Development with the jABC. In *Hardware and Software, Verification and Testing*, volume 4383 of *Lecture Notes in Computer Science*, pages 92–108. Springer Berlin / Heidelberg, 2007.