Teaching Software Modelling in an Undergraduate Introduction to Software Engineering

Bernd Westphal
Department of Computer Science
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany
westphal@informatik.uni-freiburg.de

Abstract—In this article we present our design of an (otherwise completely ordinary) undergraduate introduction to software engineering with an emphasis on contemporary software modelling.

A distinguishing aspect of our course is that we aim at a comprehensive introduction of modelling in two regards. Firstly, we introduce proper sub-languages of common modelling languages like UML class diagrams (rather than sampling examples or covering as many building blocks as possible) with a complete formal abstract syntax and semantics (so to give exact meaning to all models from the sub-language). Secondly, we emphasise issues arising from software models in the context of software engineering, e.g., that (formal) analysis results needs proper interpretation wrt. the considered software.

We discuss our objectives wrt. modelling in software engineering, and outline the content of the course and the narratives that we use to reach these objectives. Evaluation results from four seasons of teaching the course give no indication of over-straining students wrt. level or workload.

Index Terms—education, teaching, software modelling, software engineering, formal methods

I. INTRODUCTION

Modelling and model-based software development are important aspects of the discipline of software-engineering [1]. There is a large body of literature on teaching specialisation courses (or units thereof) on modelling languages such as UML [2], model-driven software development [3], modelling and modelling languages in the context of software design [4]–[6], and how to position the aspect of modelling in overall curricula [7]. In this article, we present the design of a new undergraduate course that discusses (software) modelling in the context and from the perspective of software engineering in a broader sense [8].

The need for a new design was motivated by an issue with popular textbooks on general introductions to software engineering (such as [9]–[12]). We observe that modelling languages are prominently present (in the form of class diagrams, statecharts, etc., and UML in general), yet crucial premises to an effective use of modelling in software engineering (such as (concrete and abstract) syntax, well-formedness of models, pragmatics, and (most importantly) semantics) receive (in our opinion) hardly sufficient treatment in general textbooks. We see the following instances of ‘bad modelling teaching practices’ as identified by Burgueño et al. [13]: Many examples seem to be over-simplified (cf. [14]) and tailored to the purpose to easily ‘read out’ the examples (rather than discussing models of real systems from a serious domain, or completely abstract ones), and explanations focus on syntax and intuition, and disregard a proper treatment of semantics. This approach may cause the effect that “[students] tend to see models as drawings” [15] (or: only as drawings). Lehmann & Buth [16] point out that we can hardly expect students to meaningfully solve modelling exercises on this basis, to which we would like to add that we can as well hardly expect students to meaningfully solve corresponding exam tasks.

We see two broad approaches to construct an undergraduate introduction to the very basics of software engineering with a comprehensive treatment of contemporary software modelling: To extend a software modelling course towards these basics as provided by the textbooks, or to complement the textbooks’ content with a comprehensive introduction to software modelling. In this article, we report on our efforts towards the second approach. We rely on the proven textbooks for broad concepts, vocabulary, and the ‘softer’ aspects of software engineering (like the role of humans in software engineering processes, project management, etc.), and spend the time that is otherwise used on ‘reading out’ examples on a comprehensive, formal introduction of modelling languages motivated by problems of the sub-disciplines of software engineering (such as requirements engineering, design, or code quality assurance). Our course design can be seen as a proposal to integrate the large body of research and experience from the (software) modelling community (most prominently visible on almost 15 years of the Educators Symposium) into an ‘otherwise completely ordinary’ introduction to software engineering for undergraduate students in computer science, and as an investigation of the hypothesis that this approach is feasible without sacrificing basics and without over-straining students wrt. level or workload.

The paper is structured as follows. In Section II, we describe the situation of our course in the overall curriculum and the previous knowledge of our audience. Section III elaborates on our educational objectives wrt. modelling and the overall construction ideas for the course. We outline the content and principles that we use to reach our objectives for modelling in general, and for structural and behavioural software modelling in particular, in Sections IV to VI. Sections VII and VIII discuss the complementing exercises and reports on experience from four years of teaching our course. Section IX concludes.
Note that our writing may partly switch to a didactical tone (to give an impression of our teaching); yet this article is not supposed to teach the reader new concepts or techniques. The contribution of this work is our definition of objectives, our corresponding selection and combination of content and narratives, our new definitions of modelling sub-languages for teaching, and our experience with teaching the course.

II. COURSE AND STUDENTS’ SITUATION

The course that we report on here is called ‘Software Engineering’. It is scheduled in the 4th semester of a study plan for a B. Sc. in computer science and has assigned 6 ECTS credits. The number of attendants varied over the last five seasons between about 100 and 150. About a half to two-thirds of the audience are students from the B. Sc. program. In addition, students from a non-consecutive M. Sc. in computer science and a B. Sc. and M. Sc. in Embedded Systems Engineering attend in varying proportions.

The audience tends to come with a strongly heterogeneous previous experience. In each first week of the last four seasons, we conducted a survey where we ask the students to self-assess their previous experience on a subjective Likert-scale from 0 (none at all) to 10 (3-years responsible work experience). Figure 1 shows the results of the survey from 2019 on the topic ‘design modelling’. The main properties of the data sets are mostly stable over the seasons: The minimum is 0, the first quartile is at 0 or 1, and the median at 1 or 2. The maxima were 7, 10, 9, and 10. We do present the results to the students to point out that the course is designed to take care of the part of the audience that chose value 0, that one is not alone in this part, and to invite the students who chose higher values to contribute their experience and challenge the course content.

By the schedule in the curriculum, we can assume students to have taken programming courses, introductions to algorithms and data structures, networks, operating systems, databases, and (most importantly) mathematics and theoretical computer science. Our course strongly benefits from this situation. We need not introduce many mathematical or theoretical basics, we can provide the narrative that our course can be seen as ‘use everything learned so far, if it can contribute to successful software development projects’ (including mathematics and theoretical computer science [6]), and we can draw realistic examples from the courses in applied computer science. Note that our course is the first, but not the only place of teaching software engineering topics in the curriculum. The curriculum comprises a one-semester practical course and offers specialisation courses, e.g., on UML.

III. OVERALL COURSE OBJECTIVES AND CONSTRUCTION

The compulsory objectives of the software engineering course reported on here (as defined in the module handbook) are similar to many such courses in Germany with, among others, [9] as recommended literature. Our interpretation of the module handbook adds the following objectives specific to software modelling:

1) Students are able to explain a general notion of model and to discuss software models in this context.
2) Students have basic capabilities of using (understanding, analysing, creating (to some extent)) software models.
3) Students have a broad overview of software modelling languages (including examples for different forms of concrete syntax, views, perspectives, and formality), and are aware of their advantages and limitations.
4) Students are able to discuss which (and in how far) software models address common, well-known problems and issues in the software engineering process such as misunderstandings and detecting errors late.

Our Objective 2 emphasises lower levels of Bloom’s revised taxonomy [17] than specialised courses in, e.g., model-driven software engineering do. Our focus for our general, introductory course is reading and understanding existing models, applying analysis procedures or tools to existing models, and extending (or changing) given models. Yet we do have some exercises on creating models and point out the importance of presentation and readability of models (in the sense of [18]).

With Objectives 3 and 4, we aim at the higher levels of Bloom’s revised taxonomy, namely analysis and evaluation of particular models, modelling languages, and model-based approaches in general. Objective 1 includes the concept of abstraction (cf. [4], who quote C.A.R. Hoare as “In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction.”). Overall, we emphasise concepts and principles. We do not teach in detail how to come up with a good design idea (except for very general principles such as modularity), but we teach how to ‘write down’ or describe design ideas and how to analyse descriptions of designs. Our course is after all neither a dedicated modelling language, nor modelling, nor software design course, but a general introduction to software engineering.

One principle that we apply in order to reach our objectives is ‘interpolation rather than extrapolation’. One critique on popular software engineering textbooks (as already touched upon in the introduction) is that it is common to present a selection of example models, informally ‘read out’ their meaning (or intuition), and expect students to extrapolate the (precise) meaning of the discussed model (or even modelling language). Our perception is that the discussion of software modelling often remains informal or semi-formal, with the additional information that modelling languages with a pre
cise, mathematical semantics exist, yet lie outside the scope of the book (hence the readers again need to extrapolate). We illustrate our perception in Figure 2. The double-line arrow represents our view on how many software engineering textbooks cover the formality scale. Our approach is represented by the arrow to the right: We cover informal and semi-formal techniques, and then ‘jump’ to the other extreme end of the scale. That is, we introduce some modelling languages with a completely formal abstract syntax and semantics, since semantics is a natural part of software modelling [1], [6], [19]–[21] and a necessary prerequisite for meaningful, exact, and automatic behavioural analyses. Orthogonal to the formality scale, we see a (subjective) scale of simplicity (including the effort needed to define a language, conciseness, expressive power, etc.). We discuss modelling languages from both ends of the scale. In Figure 2, each diamond represents one modelling language from our course. Our intention is to enable students to interpolate any needed level of formality or simplicity based on our sampling.

![Fig. 2. Formality and simplicity scales.](image)

Another principle we use is to teach modelling languages comprehensively. That is, we define the syntax of a manageable sub-language that is large enough to reach our teaching objectives, and small enough so that the meaning of models in the sub-language can be fully (and formally) understood. For, e.g., class and object diagrams and OCL, we define proper sub-languages that are sufficient to convey the essential potentials and difficulties of, in this case, structural modelling. We consider proper sub-languages in the sense that the content of the course that we report on here is a subset and special case of the content in our specialisation course on (a formal semantics for) UML. Our sub-languages are equipped with formal abstract syntax and semantics and thereby become comprehensive in two senses: Students are able to see the whole extent of our modelling languages and are able to fully understand (or analyse) the meaning of models up to the most intricate corner-cases. To stress our points, we offer to take the perspective of particularly challenging situations, such as completely separate customer and developer parties (not inside one company or team) negotiating a contract on requirements, or distributed, safety-critical systems, again with the intention to allow the students to interpolate the less challenging settings. We strive to show modelling languages ‘in their natural habitats’, that is, in the context of (real-world)

problems where each language is evidently good at solving problems (such as distributed, reactive systems for Statechart-like models).

From a didactical point of view, we strive for a good constructive alignment [23] between lecture content, exercises, and the exam in the sense of [24]. The process by which we constructed our course can be seen as an instance of ‘Lecture Engineering’ [16]: The course has been, to a large amount, created backwards from a form of exam that we envisioned.

Table I gives an overview of the course content. Content is grouped into four prominent topic areas of software engineering. Each topic area begins with an introduction of vocabulary and concepts. Our presentation largely follows the textbook [12] (and [25] for requirements engineering). We appreciate of this textbook how it emphasises the human aspect of software engineering, the engineering aspect (cf. [26]) and the role of modelling (yet in a mostly informal form), and its strive to be evidence-based and to relate to ISO, OMG, etc. standards if possible. At the appropriate places we complement the textbook by providing a precise syntax and semantics for some considered modelling languages. The engineering aspects of the particular topic areas (like avoiding misunderstandings in requirements engineering) directly motivate our treatment of (formal) models and exact analyses (also cf. [27]). Interestingly, our motivation of modelling languages by topic area (thus from the software engineering perspective) yields roughly the same sequencing as developed in Kuzniarz & Staron [2] from a modelling language perspective.

Though one focus of our course is formal modelling, we have chosen not to follow Bjørner’s textbook series [28]–[30] where software engineering is presented as an almost completely formal, mathematical activity. In our opinion, this approach is too far away from today’s practice of software engineering to be suitable for an undergraduate introductory course. The textbook by Mills [31] is closer to our needs yet leaves a larger gap between the concrete syntax of, e.g., class diagrams and an underlying formalisation than our approach and puts lower emphasis on putting (formal) modelling into the software engineering context.

IV. SOFTWARE MODELLING

In this section, we elaborate on our interpretations of the objectives introduced in Section III wrt. software modelling in general and outline the practices, narratives, and messages that we use towards reaching these objectives. The following Sections V and VI give more details on the sub-languages that we use for structural and behavioural models of software.

A. Modelling and Abstraction

Towards Objective 1, our course is based on a notion of model from [32], as also prominently covered in the textbook [12] that we use and in, e.g., [33]. In short, a model is an image (descriptive) or pre-image (prescriptive) of an original with three constituent properties: The image attribute, the reduction attribute, i.e., only those aspects of the original attribute, are part of the model. The reduction attribute includes parts of the original attribute that are not part of the model.
that are relevant in the modelling context are represented, and the **pragmatic attribute**, i.e., the model is built in a specific context for a specific purpose. In this sense, our notion of model is a bit broader in scope than the one considered in [1].

As food for thought, we constrain the broader definition given above by a second definition that characterises a model as an **abstract, formal, mathematical representation or description of structure or behaviour of a (software) system**. Together, we understand the term ‘software modelling’ in the narrower sense that includes precisely defined (concrete and abstract) syntax, well-formedness, and (most importantly) semantics.

In our experience, it can pose difficulties to students to ‘put life’ into such an abstract definition. Therefore, we have decided to begin with modelling as such and postpone the definition a bit. We start to introduce models in the topic area ‘Project Management’ in the form of (semi-formal) process models (built of roles, activities, artefacts, etc. as in the V-model) and point out how a concrete process can be abstracted or generalised into a process model (descriptive) and how a process model can prescribe and give rise to a concrete sequence of activities in a concrete project. During this time, we pay close attention to using the word model consistently with the definition that we are to present later.

Formal methods appear in the topic area ‘Requirements Engineering’. Motivated by the ubiquitous problem of imprecision and misunderstandings, we focus on how formal modelling languages (with syntax and semantics) can help to improve precision. To this end, we discuss an as-simple-as-possible but not trivial modelling language, in our case Decision Tables (as presented in [10], [11], with the real-world correspondence of business rules [34]). Already at this place we point out that the choice of **observables** (conditions and actions in the case of Decision Tables) is a creative act (of modelling). And we prominently discuss semantic variation points (cf. [35]) by introducing two semantics for Decision Tables. In our opinion, it is necessary to make the students aware of the fact that it is not uncommon in today’s practice that different modelling communities or tools (cf. [19] for the case of OCL) assign different meanings to one and the same model. For details on the topic area ‘Requirements Engineering’ we refer the reader to [36].

The definitions given above appear early in the topic area ‘Design & Architecture’. We draw on the analogy to floor-plans in construction engineering (cf. [37], [38]). A floor-plan abstracts from certain details and allows the engineer to analyse certain requirements (such as “the bathroom needs to have a window”) already on the plan. We point out that this analogy does not carry arbitrarily far: Firstly, in particular floor-plans of family houses are much simpler than software (hence the plan of a large hospital or a plan from regional planning would be a slightly better example). Secondly, floor-plans only have a very weak **behavioural aspect** (if we want to, we can read the area of movement of doors as behaviour that can be analysed for the absence of collisions). Still, in our experience, the usefulness of the analogy is stronger than the possible confusion for being basically a structural model. At this point in the lecture, we point out that we already discussed examples of modelling in the course: In the form of process models and in the topic area requirements engineering.

One of the central messages of our course is the topic of abstraction in software engineering and how software models are abstractions, last but not least from code (cf. [4] and the discussion of Objective 1 in Section III above). Because of the construction of the curriculum into which our course is embedded, we may and do deliberately, demonstratively, and explicitly not discuss code during the first half of the course. In the very first lecture, we state that we view this course to address software engineering in the sense of ‘everything that lives around programming’ (programming is part of software engineering in general but not of our course, and well covered in other courses). This approach is supported by the sequence of topic areas in our course: Students understand that requirements engineering can be seen as only indirectly being about code and programming, but first of all about understanding and analysing system requirements. We point out how the latter can be supported by (formal) models. In the topic area ‘Design & Architecture’, we explicitly draw the connection to, e.g., class diagrams as visualisations (descriptions, images) of existing code, which most students have seen in their previous courses, and state that in this course we are after a more abstract view and use of class diagrams. A view where class diagrams **may** later be implemented in an object-oriented programming languages (manually or via code-generation), but that this is not the only purpose that class diagrams can serve.

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic Area</th>
<th>Content Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>Software, Engineering, Software Engineering; Successful software development, empirical data on project success.</td>
</tr>
<tr>
<td>2-5</td>
<td>Project Management</td>
<td>Software metrics, scales; Cost estimation (experts’ and algorithmic estimation); Project, process, process modelling; Procedure models; Process models (agile, V-Model (semi-formal)).</td>
</tr>
<tr>
<td>6-10</td>
<td>Requirements Engineering</td>
<td>Requirements, Requirements Analysis; requirements properties (completeness, consistency, etc.); kinds of requirements (tacit etc.); Dictionary; Requirements specification languages: Natural language patterns, Decision Tables (formal), Use Cases and Use Case Diagrams (semi-formal), Live Sequence Charts (formal).</td>
</tr>
<tr>
<td>11-14</td>
<td>Design &amp; Architecture</td>
<td>Model, Views; Structure Models (Class and Object Diagrams (formal), Proto-OCL (formal)); Behaviour Models (Communicating Finite Automata (formal), Query Language (formal)); an outlook on UML state machines; design principles and Design Patterns; model-based/model-driven software engineering (MBSE/MDSE).</td>
</tr>
<tr>
<td>15-18</td>
<td>Code Quality Assurance</td>
<td>Test case, test execution, true/false positive/negative outcomes (formal); Limits of testing; Glass-box testing (statement, branch, term coverage); Model-based testing and runtime-verification; Program verification (formal), Code review.</td>
</tr>
</tbody>
</table>
### B. Modelling Languages

Our approach towards Objectives 2 and 3 is to choose the modelling languages that we discuss in the course in a way that a broad range of characteristics of modelling languages is covered. Table II shows a selection of well-known aspects [1] of modelling languages. We see that we cover a range from simple to complex modelling languages (Live Sequence Charts [39], [40] as the example for a complex one), graphical and textual concrete syntax, structural and behavioural views, modelling languages that lend more to reflective (the ‘what’) descriptions and ones for more constructive (the ‘how’) [41], and the complete formality scale from informal (we also discuss natural language in the topic area requirements engineering), over semi-formal, to formal. In the topic area ‘Design & Architecture’, we have intentionally chosen one graphical and one textual and one reflective and one constructive modelling language for the two views of structure and behaviour to emphasise the orthogonality between the views and available modelling languages. We elaborate on the chosen (sub-)languages in Sections V and VI.

#### C. Models in the Context of Software Engineering

From the software engineering perspective and towards Objectives 3 and 4, an important topic is how models and analysis results relate to the modelled software. That is, which conclusions for the final system can be drawn if a model-checker reports that a certain invariant holds for a behavioural model of our design. We propose to call the outcome ‘invariant holds’ a negative: The analysis for the presence of an error has been negative (as in test theory). Then each positive or negative outcome can be ‘true’ or ‘false’: The outcome is a true negative if the considered design does satisfy the invariant. The, in our experience, most difficult to grasp case is the false negative. We point out that false negatives can have manifold reasons: Ranging from invalid models (e.g. due to ‘typos’ when writing the model) to errors in the analysis tool. The positive cases are easier to understand (and to deal with): Many analysis procedures produce counterexamples if, e.g., an invariant is found not to hold. This counterexample can be discussed with the designers to determine whether there is a true or false positive. Overall, we can conclude that a (formal) analysis of a design model with a negative outcome allows to conclude that, if the model is valid, and if the analysis tool is correct, and if the environment assumptions of the model hold etc., then the system will not fail due to a design error (thus broadly following Jackson’s proposal of dependability cases [42]). Without a thorough analysis of a design idea, an observed system failure may be due to a design flaw (as well as due to an error in the implementation, or the compiler, or the operating system, or the hardware). Thus one can take the view that the effort investigated in modelling and analysis comes with the gain of lowering the overall risk for errors in the final system (next to all other well-known possible benefits of software modelling [43]). Whether and which effort (and which modelling language) is appropriate for a given problem in a given development project, depends on the context; the appropriate effort may be much higher for safety-, mission- or business-critical features of a software than for uncritical convenience features.

The topic of model validation raised in the previous paragraph is in our perception closely related to a misconception about (formal) software modelling that is prominently found in software engineering textbooks (even in our highly appreciated [12]) and that re-appears in personal communication with practitioners. Colloquially put, “the clients do not understand (formal) models”. Our position is that software models are first of all tools ‘for us software engineers’: To increase precision and clarity about design decisions, to support the cooperation of teams, to provide means for thorough analyses. In the lectures, we use the analogy of individual contracts created by lawyers. Persons who have not studied law can not be assumed to understand the implications of an involved individual contract because it is written in the language of law. An individual contract needs to be validated by scenarios. That is, the client names scenarios with expected outcomes and the lawyer is supposed to determine the outcome defined by the contract. If the outcomes do not match, the contract needs to be reworked. We propose the same approach for software models: To validate a behavioural model, we can ask the designers (who may only have a background in communication or electrical engineering) about scenarios and expected outcomes and then the software engineer replays the scenario in the model and interprets the outcome for the designers.

To conclude this section, we would like to stress that we (in the spirit of [12]) in particular do not advocate for or against formal modelling in general. We point out that there

<table>
<thead>
<tr>
<th>Process Model</th>
<th>simplicity</th>
<th>concrete syntax</th>
<th>view</th>
<th>perspective</th>
<th>formality</th>
<th>exercises</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision Tables</td>
<td>sub-language</td>
<td>graphical</td>
<td>process</td>
<td>constructive</td>
<td>semi-formal</td>
<td>paper</td>
</tr>
<tr>
<td>Use Case Diagrams</td>
<td>simple</td>
<td>tabular</td>
<td>behaviour</td>
<td>reflective</td>
<td>formal</td>
<td>paper</td>
</tr>
<tr>
<td>Live Sequence Charts</td>
<td>complex</td>
<td>graphical</td>
<td>behaviour</td>
<td>reflective</td>
<td>informal</td>
<td>paper</td>
</tr>
<tr>
<td>Class and Object Diagrams</td>
<td>sub-language</td>
<td>graphical</td>
<td>structure</td>
<td>constructive</td>
<td>formal</td>
<td>paper</td>
</tr>
<tr>
<td>Object Constraint Language</td>
<td>sub-language</td>
<td>textual</td>
<td>structure</td>
<td>reflective</td>
<td>formal</td>
<td>paper</td>
</tr>
<tr>
<td>Communicating Finite Automata</td>
<td>medium</td>
<td>graphical</td>
<td>behaviour</td>
<td>constructive</td>
<td>formal</td>
<td>tool</td>
</tr>
<tr>
<td>Query Language</td>
<td>medium</td>
<td>textual</td>
<td>behaviour</td>
<td>reflective</td>
<td>formal</td>
<td>tool</td>
</tr>
<tr>
<td>Statemachines</td>
<td>sub-language</td>
<td>graphical</td>
<td>behaviour</td>
<td>constructive</td>
<td>informal</td>
<td>-</td>
</tr>
<tr>
<td>Pre-/Post-Conditions</td>
<td>medium</td>
<td>textual</td>
<td>behaviour</td>
<td>reflective</td>
<td>formal</td>
<td>paper &amp; tool</td>
</tr>
</tbody>
</table>

#### TABLE II

**CLASSIFICATION OF MODELLING LANGUAGES DISCUSSED IN THE COURSE.**
are software engineering problems to which (to the best of our knowledge) formal modelling and analysis has been and can be a (or: the state-of-the-art) solution. In other words, if there is an issue that can be solved with a model in a particular modelling formalism, then one should consider using this approach. And if a model is not needed (if there is no purpose), then it should not be used. In addition, our course explicitly appreciates and points out the value of informal models and sketches in software engineering (cf. [13], [44]–[48]). We share the view that engineers who are aware of the precise syntax and semantics of the modelling formalism that is used for sketching can work much more effectively because each participant in the discussion would be able to ‘fill in’ the necessary precision should it come to disputes over the meaning of the sketch.

We strongly agree with the observation of Paige et al. [20] that ‘providing answers, not solutions’ is a bad practice. Yet in our course, we need to take a differentiated view. Since we provide a precisely defined syntax and semantics for most of the modelling languages discussed in the course, there is certainly a clear ‘wrong’ or ‘right’ wrt. questions regarding the syntax or the formal meaning of a given model. And the students appreciate this clarity a lot, both in the exercises and in the exam review. Whether a model is a good solution can be subject to discussion and opinions. Therefore, we cautiously avoid to give the impression that, e.g., the proposals discussed in tutorials are (the one and only) ‘reference solution’ (cf. Section VII).

V. Structural Software Modelling

In the popular textbooks on software engineering, the predominant modelling languages for structural aspects of software are class and object diagrams from the Unified Modelling Language [49], and (to some extent) the Object Constraint Language (OCL) [50]. Therefore, our course uses these three as examples of structural modelling languages.

Our objectives wrt. structural modelling are the following. We want to offer the view that class diagrams are (also but not necessarily) visualisations (or pre-images) of object-oriented programs. Class diagrams are also used to model, e.g., problem domains, databases, or distributed systems. We want to show how class and object diagrams are formally connected, how object diagrams can be used to illustrate the intended use of a data-structure (modelled by a class diagram), and how object diagrams can be used for analysis purposes to infer needed data-structures. An objective wrt. OCL is to point out that OCL is a three-valued logic, i.e., that OCL constraints can evaluate to ‘true’, ‘false’, or ‘undefined’. The concept of three-valued logics is not familiar to the majority of our audience, so we point out how the value ‘undefined’ may arise in object structures. From a software engineering perspective, we point out that if an OCL constraint is meant as an invariant, then one should read the constraint as the requirement that the constraint always evaluates to ‘true’ on all system computations (and not that the constraint should never evaluate to ‘false’, which is equivalent in two-valued logics).

In the following, we outline our choice of UML sub-languages and the formal semantics that we use in our course. The choices are based on our work and experience in the area of software modelling (e.g. [51]–[56]) and of many years of teaching a specialisation course on UML (with a strong emphasis on formal semantics). Our background certainly influences our teaching. Yet in the design of the software engineering course that we report on here, we have taken extensive care to keep the course self-contained and independent from a particular person. The mathematics used for the formal semantics are intentionally kept as basic as possible to open the long-term perspective to integrate the new formalisations presented in this article into a general purpose textbook to be used by teachers who need not have a that strong background in MDSE and formal methods.

A. Class and Object Diagrams

The formal class diagram language that is sufficient for our purposes is (astonishingly?) small. We mostly discuss classes with attributes and methods, and binary associations with one navigable, owned end and one non-navigable end. Multiplicities are limited to 0..1 and 0..*. Figure 3 shows a typical class diagram from the lecture, on top with the necessary decorations from the UML standard, and on the bottom in an abbreviated form agreed upon in the lecture (we may leave out ownership and non-navigation decorations).

The abstract syntax (or signature) of a class diagram is simply a tuple \( \mathcal{D} \) including a set of basic types (such as \( \text{Int} \)), a set of classes, a set of typed attributes of basic type or derived type \( C_{0,1} \) or \( C_{*} \) (where \( C \) is a class), and a mapping from classes to attributes. The derived types \( C_{0,1} \) and \( C_{*} \) are used to demonstrate the different treatment of objects and collections in OCL (see below). A structure \( \mathcal{D} \) assigns domains (or values) to the basic types and provides an infinite set of object identities for each class; inheritance can be treated by allowing non-disjoint sets of object identities. Given a structure, the meaning or semantics of a class diagram is the set of system states over \( \mathcal{D} \) and \( \mathcal{D} \), thus we use a simplification of the formal semantics provided in the appendix of the OCL standard document [50]. A system state \( \sigma \) is a partial function from object identities (the objects in the domain of \( \sigma \) are called alive) to a type-consistent mapping of attributes (of the corresponding class) to values. That is, a system state assigns to each alive object an attribute-value mapping. A system state can be visualised by an object diagram.

This particularly small class diagram language is sufficient to reach our objectives: We re-emphasise the difference
between concrete and abstract syntax (that abstract syntax in particular captures the semantically relevant information from a concrete syntax, and that there is a many-to-one relation between well-formed diagrams and abstract syntax), we discuss the importance of layout and presentation for readability (our sub-language allows to construct arbitrarily unreadable class diagrams), we present the uses of object diagrams (from illustration of intended data-structure use (as a simple alternative to completely worked out OCL constraints) to analysis (infer class diagrams from scenario object diagrams)). We point out the difference between partial and complete object diagrams (in contrast to system states, where each object must have a value for each of its classes attributes), and we demonstrate the intricacies of OCL that stem from the possibility of not defined or ‘dangling’ links. In addition, the formal semantics is a proper simplification of content from the standard document [50].

Note that we do not advocate to not expose students to a much richer feature set of UML class diagrams. Our choice is driven by the particular objectives of our course and feasible since, by the position of our course in the overall curriculum, we can assume that the students know from previous courses (programming, algorithms and data-structures) that the UML class diagrams language is much larger.

B. Proto-OCL

We introduce OCL as an example for a textual, reflective structural modelling language. Our main objective is to outline how OCL expressions can be evaluated on object diagrams (and thus be used to formalise invariants or pre- and post-conditions), and that OCL is a three-valued logic (see above).

For didactical reasons, we introduce Proto-OCL, which is a sub-language of OCL in a concrete syntax that resembles the classical first-order logic that the students learn in previous courses. In our experience from our specialisation course on UML, it takes the students some time to get used to the concrete syntax of OCL. Our concrete syntax avoids this additional cognitive load, is sufficient for our objectives, and we of course discuss the relation between our syntax and the one from the standard documents, we only do not work formally with the latter.

The syntax of Proto-OCL is given by the typed grammar in Figure 4. Here, c is a logical variable, C a class from \( \mathcal{C} \), \( v_1 \), \( v_2 \), and \( v_3 \) are basic type, \( D_{0,1} \), and \( D_k \) attributes of C, respectively, and, e.g., \( \tau_C \) is the OCL (!) type of object identities of \( C \), including the value ‘undefined’ (or \( \bot \)). An example expression (over the class diagram from Figure 3) is \( \forall d \in \text{allInstances}_{D} \bullet x(d) \geq 0 \), which corresponds to context \( d : D \text{ inv} : d.x \geq 0 \) in OCL. The semantics of Proto-OCL is defined inductively, following [50]. Overall, the definition of Proto-OCL fits onto three lecture slides which gives us (space and) time to introduce the definitions slowly and carefully, and to elaborate on our objectives.

VI. BEHAVIOURAL SOFTWARE MODELLING

As behavioural modelling languages, the course presents Live Sequence Charts (LSCs), Communicating Finite Automata (CFA), and a so-called Query Language. Of LSCs [39], we present almost the complete syntax (basically disregarding cold messages) and the original automaton-based semantics [40]. Our main objective is to show that not all (graphical) modelling languages are as small as our sub-languages, but that more concise and expressive languages need more work to define and understand the syntax and semantics. LSCs naturally appear in the topic area ‘Requirements Engineering’, where we discuss (informal, textual) use cases [57], semiformal Use Case Diagrams, and complete the picture with the formal sequence diagram dialect LSCs.

In the topic area ‘Design & Architecture’, we focus on a Statemachine-like modelling sub-language with tool support. Our main objective on a conceptional level is to connect the structural and the behavioural view, and, from a software engineering perspective, the uses of model simulation and verification (cf. [13], [47]). The language that we use are the timed automata of the tool Uppaal [58] without time; we call the language communicating finite automata (CFA). Formally, a CFA consists of a finite set of locations with exactly one initial location, a set of channels, a set of variables, and a finite set of edges. An edge is a 5-tuple with source and destination location and an action (input or output on channel, internal action \( \tau \)), a guard over global and local variables, and a finite update vector. CFA come with a rich, Java-like basic type system and expression language. The formal semantics of a finite set of CFA (a network of CFA) is a labelled transition system over configurations of the form \( \langle \vec{\ell}, \nu \rangle \) where \( \vec{\ell} \) is the current location vector (one for each CFA in the network) and \( \nu \) a valuation of the variables (cf. [59]). There are two kinds of transitions: There is an internal transitions if a \( \tau \)-edge is enabled in some CFA in the network, and a synchronisation transition if two edges with input and output actions on the same channel are enabled in two different CFA.

We have chosen CFA for their simplicity and tool support. The formal definitions of the abstract syntax and the semantics fit onto five slides. Yet CFA are sufficient for our objectives. They allow us to point out the powerful abstraction means of non-determinism, both internally and as interleaving of transitions, and the difficulty of spotting errors in a distributed system, even for quite small models.

We particularly like the tool Uppaal for teaching purposes due to its remarkably reduced user interface (cf. Figure 5, as compared to state-of-the-art UML tools like IBM Rhapsody, which we use in our specialisation course on UML). Uppaal comes with a powerful simulator which particularly supports our teaching goals wrt. views and behavioural modelling and

\[
F ::= c : \tau_C \mid \text{allInstances}_C : 2^{\tau_C} \mid v_1(F) : \tau_C \rightarrow \tau_\bot \mid v_2(F) : \tau_C \rightarrow \tau_D \mid v_3(F) : \tau_C \rightarrow 2^{\tau_D} \mid f(F_1, \ldots, F_n) : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \forall c \in F_1 \bullet F_2 : \tau_C \times 2^{\tau_C} \times B_{\bot} \rightarrow B_{\bot}
\]

Fig. 4. Proto-OCL formulae over signature \( \mathcal{C} \).
Compared to UML Statemachines, CFA are simpler in that they do not support hierarchical states and connectors, and that the synchronous communication is much easier to define and explain than the asynchronous run-to-completion semantics of UML Statemachines. Note that CFA hardly qualify as object-oriented, hence the underlying semantical models of class diagrams and CFA are not formally integrated. We welcome this separation since, in our opinion, it supports our overall objective to teach the concepts of software modelling: Our point is that it can be useful to model structure using class diagrams even if the behavioural model (or the final implementation) is not based on an object-oriented language. That is, we intentionally confront the students with this gap (in contrast to, e.g., [6]), yet we do have a continuously used, highly abstract definition of software as a description of a set of computation paths that conceptionally integrates all formally discussed modelling languages.

Introducing CFAs (including a Uppaal demo), the Query Language of Uppaal (which is a tiny fragment of the temporal logic CTL), and the uses of the verifier on networks of CFA and queries for drive-to-configuration or scenario-verification analysis. Using Figure 5, we explain that in the top-right and middle area, we see the current configurations (locations and variables), which can be seen to correspond to an object diagram. The top-left area shows the transitions enabled in the current configuration (in Figure 5, we have run into a deadlock), and the bottom-left area successively constructs a computation path based on the chosen transitions. As the ‘icing on the cake’, Uppaal also represents the computation path as a sequence diagram, so that we can draw the connection to our study of LSCs earlier in the course, here used in a descriptive manner. Note that we present Uppaal from a pure user’s point of view: We report that the model-checking problem for CFAs is decidable and complex, yet we do not discuss model-checking algorithms.

A prominent discussion in teaching software engineering and modelling is the use of tools vs. paper & pencil exercises (cf. [3], [13], [20], [47], [60]). Given our objectives, the analysis of Glinz [61] applies and thus the majority of our exercises is in paper & pencil form (cf. Table II). One of our objectives is that the students first understand the modelling languages to then be able to appreciate and assess aids provided by tools. Given the comprehensiveness of our sub-languages, we consider the effort for learning a tool for, e.g., class diagrams or LSCs, too high compared to our goals. For behavioural modelling, tool support is, in our opinion, nearly a must (cf. [13], [47]) since a tool with simulation (and verification) support can make the benefit of behavioural models ‘tangible’. In our experience, a complete analysis of behavioural models of, e.g., distributed systems is almost immediately out of scope for the human mind, and we want to confront our students with this experience. Therefore we, as many others, use tools for the analysis exercises.

Our exercises and tutorials are constructed according to the following principles. The exercises are about half technical (‘learn for the exam’) and half open (‘learn for life’). In the tutorials, the submitted solutions are used for discussions, e.g., of the quality of a submitted model, and for further questions, e.g., how an analysis result needs to be interpreted in a software engineering project, all to practice the human aspect of software engineering. Our student tutors assume the role of moderators: They do not present one ‘reference solution’, but the audience develops one good solution together and the tutors moderate the discussions and take notes using a pen-enabled screen.

With our students we can confirm the observation stated in [4], that the participants are mostly not used to imprecisely stated exercise tasks. In our opinion, software engineering exercises need to (at least to some amount) cater for the inherent imprecision and openness of software engineering tasks, hence we take a two-step approach to the design of exercises:

**Fig. 5. Demo of the Uppaal simulator.**
Firstly, we sketch an intended solution and grading scheme, and secondly we develop a task description with an amount of imprecision that is appropriate to the undergraduate audience (in our graduate level specialisation course, in contrast, much higher degrees of imprecision are possible).

VIII. Evaluation and Experience

The most pressing question in the first season of our investigation of the hypothesis that a formal and comprehensive introduction of software modelling is possible in an undergraduate introduction to software engineering was whether we overstrain the students wrt. level or workload. Figure 6 shows results from the regular, subjective students evaluation of the previous four years (with 26 to 39 responses). We are perfectly happy with the outcome: In our opinion, our courses should be challenging but not overstraining. Figure 6 also shows that we have succeeded in fine-tuning the course so that the response ‘too high’ does not occur any more. We are also happy with the overall feedback (not reported here, cf. [62]), and are perfectly satisfied (as well as the students) with the (remarkably stable) distribution of grades.

It is notoriously difficult to evaluate the effect of our course on the future work situation of the students. As a start, colleagues report that the course provides a good common ground to discuss advanced thesis topics. On our presentation of the requirements engineering [36] and the German [63] perspective of our course, we have received positive feedback and industry practitioners (from aerospace, automotive, automotive suppliers, internet commerce, etc., to small embedded systems companies) encouraged us in personal communication to pursue our approach, often reporting that the use of (formal) models is desired but not sufficiently supported by the educational background of available personnel.

We would like to report on two observations over the seasons that may contribute to the overall data on teaching modelling. Firstly, we do not have any indications of ‘mathphobia’ that is often associated with formal methods. Our hypothesis on a reason is that we use mathematics purely problem-oriented and show that even the most basic mathematical concepts are useful to clarify syntax and semantics questions. Our impression, on the contrary, is that our students appreciate the formal treatment of modelling languages and the presence of a clear ‘wrong’ or ‘right’ wrt. syntactical or (technical) semantical questions. It puts us (and the students) into the comfortable situation that we do not observe any ‘bargaining’ on technical tasks in exam reviews: Either we together prove the solution in question wrong (and the student accepts the mark) or the student proves the solution correct (and we change the mark and apologise for our mistake). Secondly, we do not observe any indications of students missing code (cf. [3], [64] (on MDSE) vs. [20], [65]). In the first season, we (also) had the assumption that students would love coding, yet the exercise on implementing a given model was the one with the by far lowest submission rate, so we do not have any coding exercises any more.

IX. Conclusion

We have reported on our effort to integrate state-of-the-art software modelling education into an ‘otherwise completely ordinary’ undergraduate introduction to software engineering.

Our overall objective is to thoroughly teach the fundamental concepts underlying the use of (formal) software models and to emphasise the software engineering context. New aspects of our approach are the aim for ‘interpolation’ (e.g., by fully covering the formality scale with examples), and for comprehensiveness in the sense that we trade the breadth of treatment of modelling languages for an in-depth (syntax and semantics) discussion of proper sub-languages (with outlooks on the extension of the ‘full versions’). To this end, we have presented new definitions of sub-languages and argued how they are sufficient to reach our teaching objectives.

Empirical results show that our course does not overstrain our students wrt. level or workload.

Acknowledgements

We thank S. Feo Arenis and Ch. Schilling for their invaluable support as teaching assistants during the first four seasons, and our student assistants for their committed work.

References
