On Complementing an Undergraduate Software Engineering Course with Formal Methods

Bernd Westphal
Department for Software Engineering
Albert-Ludwigs-Universität Freiburg, Germany
Freiburg, Germany
0000-0002-6824-0567

***Abstract*—Software systems continue to pervade day-to-day life and so it becomes increasingly important to ensure the dependability, safety, and security of software. One approach to this end can be summarised under the broad term of formal methods, i.e., the formal analysis of requirements, software models, or programs. Formal methods in this sense are today used in many branches of the software industry, such as the huge internet companies, aerospace, automotive, etc. and even made their way into small to medium sized enterprises.**

In this article, we argue the opinion that today's students (and tomorrow's engineers) need to be provided with a basic understanding of formal methods in the broad sense (what is it, how does it feel to use it, what are advantages and limitations) already in undergraduate introductions to software engineering. We propose a generic course design that complements (otherwise completely ordinary) undergraduate introductions to software engineering with formal semantics and analyses of (visual) software description languages. We report on five years of teaching an implementation of the course design that indicate the feasibility of teaching without sacrificing classical software engineering topics and without over-straining students wrt. level or workload.

***Index Terms*—Teaching, Formal Methods, Software Engineering**

I. Introduction

Aspects such as dependability, safety, and security are of increasing importance to more and more software development projects as seen, e.g., in the extended coverage of these topics that came with the current, 10th edition of [1]. A set of methods, techniques, and tools that allow us to increase our confidence in the dependability, safety, or security of software are covered by the broad term of *formal methods*.

As the term 'formal methods' has a very wide variety of broader or narrower meanings, we begin with a clarification of our understanding for the scope of this article (and the course we later report on). Parnas [2], for example, seems to work with a very narrow definition that basically uses the term 'formal methods' as a synonym for (deductive) program verification and notations such as Z [3], Event-B [4], etc. A similarly narrow understanding is followed by the 'Formal Methods Teaching' (FMTea) Workshop series [5]. The following, much broader view was proposed by Gries [6] already in 1996: "More liberally, any informal use of theoretical ideas in the development process can be viewed as an application of formal methods." We fully agree on this view in general, yet for didactical reasons we adopt a slightly narrower understanding that excludes informal sketches. The definition that best matches our understanding of the term 'formal methods' was proposed by Bjørner & Havelund [7] (also cf. [8]):

> "A method is called *formal method* if and only if its techniques and tools can be explained in *mathematics*."

This definition includes, among others, requirements formalisation (in any mathematical form) and analysis, formal modelling (Statecharts [9], [10] as well as Z and Event-B) and model-checking, and (deductive) program verification.

In our perception, the interest, acceptance, use, and need for formal methods (in the sense of Bjørner & Havelund) in the software industry has rapidly changed in recent years.[1] In the last 10 years, the big internet companies have all established departments that focus on formal analysis of software systems, companies from the aerospace [11] and automotive [12] domain formalise and analyse requirements, and even small companies benefited from formal design models complementing their development process [13]. To best prepare our students of today for their future career in industry or academia, we feel the need to provide our students with a basic and solid understanding of the concept of formal methods, their use, and an overview over advantages and disadvantages. We should, in our opinion, aim at bridging the gap from engineering to formal methods and to this end, we investigate the complementation of existing software engineering courses with aspects of formal methods.

In this article, we propose new learning objectives regarding formal methods, and we present and discuss an approach to complement undergraduate introductions to software engineering with a comprehensive introduction to formal methods. The basic idea is to offer experience with formal methods as such on an as-simple-as-possible (but not trivial) formalism, and to complete all topic (or knowledge) areas with fully defined sub-languages of formal software description languages.

In [14] and [15], we have presented the formal sub-languages that we propose to use in the topic areas Requirements Engineering and Design, respectively, and in [16]

[1] The textbook [1], for example, bases its discussion of formal methods only on references that appeared between 1990 and 2013 (with a median of 2009, and an upper quartile of 2010).

a country-specific discussion from the software engineering perspective. In this article, we present an overall approach to the complementation of software engineering courses with formal methods in four topic areas.

In this article, we address the following research questions:

- RQ1 What are appropriate learning objectives for comprehensive teaching of the cross-cutting topic of formal methods in software engineering?
- RQ2 Is it possible (for teachers) to fit content into a one-semester course so that the learning objectives mentioned above are covered without sacrificing established, classical learning objectives for software engineering courses?
- RQ3 Do courses as named in (RQ2) necessarily overstrain students wrt. level or workload?

Towards (RQ1), we propose and thoroughly discuss three guiding learning objectives in Section II. We address (RQ2) in Sections III and IV. In the former section, we outline a generic approach (a 'design' in software engineering terms) to complement introductory courses to software engineering with formal methods. The latter section elaborates on our implementation of the design from Section III. In Section V, we report on five seasons of teaching an undergraduate introduction to software engineering followed by Section IV towards (RQ3).

II. Learning Objectives

We propose the Learning Objectives (O1) to (O3) shown in Table I to provide an understanding of the concept of formal methods, their use, and an overview over advantages and disadvantages. From the perspective of a job description, we do not aim to educate verification engineers (we leave that education to dedicated formal methods courses or curricula), but computer scientists who know the concepts and vocabulary to effectively join work situations where formal methods (simple or sophisticated) are used (or about to be used) in activities such as requirements engineering, software design, or code quality assurance.

Our learning objectives can be seen as addressing different levels of the revised Bloom's taxonomy [17]. Learning Objective (O3) addresses the higher levels of 'analyse' and 'evaluate', and basically motivates the other two learning objectives as pre-requisites. Learning Objective (O2) addresses competences at the levels 'understand' and 'apply', and Learning Objective (O1) can be seen as mostly addressing the competence levels 'know' and 'remember'. In the following Sections II-A to II-C, we elaborate on our rationales behind and our interpretations of learning objectives (O1) to (O3) in inverse order, beginning with (O3).

A. Learning Objective (O3): The Software Engineering Context

Learning Objective (O3) emphasises the (software) engineering aspect and offers to motivate the discussion of formal methods from the context (like [19]) and related to common problems in the discipline such as misunderstandings (due to imprecision) or a need to build reliable systems economically

TABLE I
Forma methods-specific learning objectives.

O1	Students have a broad overview how the term 'formal methods' can be understood in the context of software engineering and know examples of such methods.
O2	Students have basic capabilities of using (understanding, analysing, and (to a certain amount, cf. [18]) creating) formal specifications of requirements and designs, and apply program verification.
O3	Students are able to discuss which and in how far formal methods address common, well-known problems and issues in the software engineering process such as misunderstandings and detecting errors late, and are aware of the advantages and limitations of formal methods.

(cf. [20]–[22]). Our understanding of Learning Objective (O3) (in contrast to dedicated formal methods courses) is to focus on the following relations between formal methods and 'real-world software engineering' as illustrated in Figure 1 (also cf. Section 'Pragmatics' in [8]):

- Relation (1): Engineers need to well understand formal software description languages (i.e., their syntax and semantics) and relevant properties to effectively analyse formal descriptions for properties.
- Relation (2): Software engineering projects typically include the client role. Clients may need to participate in the validation of formal documents (with support from the engineers in the project) and clients may need to take decisions based on analysis results presented by the engineers.
- Relation (3): Software engineers may employ algorithms and tools to analyse formal descriptions for relevant properties. From the software engineering perspective, we consider it sufficient to view analysis algorithms and tools as 'black boxes' (where dedicated formal methods courses may emphasise the study of methods and algorithms (their correctness, completeness, complexity results, and of course the proofs thereof)). To us, Learning Objective (O3) only requires that we *report* the correctness and complexity results from formal methods research (as the specification of the black boxes).
- Relation (4): The *interpretation* of analysis results in a particular product context is of high importance to us. That is, what does a particular outcome of a formal analysis imply for the considered product, and which actions should the engineer take on that outcome?
- Relation (5): Today's practice of software engineering (as reflected in contemporary software engineering textbooks) uses a very broad variety of approaches and techniques, of which formal methods are only a small fragment. We need to explain how the course content relates to today's practice.

B. Learning Objective (O2): Value Reading, Understanding, Analyses, and Changes over Creation

To see how formal methods apply to certain software engineering problems, students should be able to use (sub-

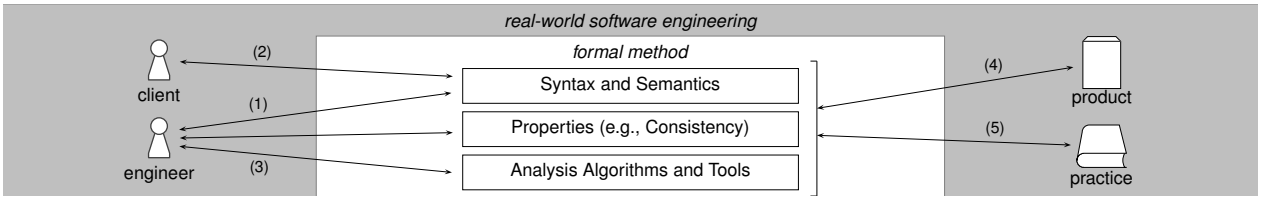


Fig. 1. Relations between three aspects of formal methods and a simplified software engineering context. (1) Engineers use formal software description languages (with syntax and semantics) to formalise aspects of software. (2) Formal software description languages need to be validated with (not by) the client. (3) Engineers may use tools to analyse formal software descriptions for properties. (4) There are relations between software products and analysis results for formal software descriptions. (5) Formal methods are related to the software development practice as described in textbooks.

languages of) formal methods to actually solve these problems. An important implication of Learning Objective (O2) is the activity of *modelling* to be conducted by software engineers. That is, the process of developing a formal model of requirements, structure or behaviour of software as a creative act. Software engineers may also need to *validate* proposed formalisations, in particular in cooperation with clients.

To this end, Learning Objective (O2) calls for a comprehensive introduction of the formalisms used in a course. It is in our opinion not sufficient to give a few example formalisations and 'read out' their intuitive meaning (as often seen in textbooks such as [1], [23], [24]), but it is necessary to define the concrete and abstract syntax, and the semantics of (possibly a subset of) a considered formalism in order to enable students to meaningfully use these formalisms. In our opinion, students need to be able to *argue* the correctness of their exercise or exam solution and to be able to follow an argument for the incorrectness of their solution in discussions with, e.g., their tutor on a solid basis for those tasks that have a well-defined notion of correctness (also cf. [25]).

There are two distinctive aspects compared to dedicated formal methods courses. Firstly, Learning Objective (O2) recommends to discuss *multiple*, possibly very different formal methods for different purposes. Secondly, many dedicated formal methods courses aim to enable their students to *create* complex formal models on their own. This is the highest competency level in the revised Bloom's taxonomy [17], and should be appropriate for a dedicated one-semester course. Learning Objective (O2), in contrast, intentionally aims lower because we have the impression that few computer science graduates are immediately confronted with the task to create formal models 'from scratch'. We hence propose to emphasise *reading* and understanding formal models, and the capability to analyse given formal models and interpret the analysis outcome. In our opinion, it is sufficient if students have *some* experience in creating formal descriptions of software aspects, yet possibly with a simple formal description languages and not necessarily with all formalisms discussed in a course.

C. Learning Objective (O1): Issues with the Term 'Formal Methods' and Consistency with Today's Practice

Given the confusion that often comes with the term 'formal methods' as such (cf. Section I), we see a need to inform students of the existence of this confusion. To enable students to resolve possible confusions with fellow software engineers, Learning Objective (O1) implies a coverage of different formal methods. Our definition of 'formal methods' as given in Section I includes very simple approaches (such as using values of well-defined metrics to specify software quality) as well as visual formalisms such as sequence diagrams or Statecharts with complex syntax and semantics. In our perception, using formal methods can be a gradual process (that is perfectly consistent with textbook recommendations) and need not be a radical change. For example, if Class Diagrams are used as sketches today, it may be a gradual step to agree on a precise concrete syntax, and then on a formal semantics that enables formal analyses of the (then formal) Class Diagram model.

III. Course Construction

In this section, we outline a generic approach to construct an introduction to software engineering that considers Learning Objectives (O1) to (O3). The idea is to take a 'completely ordinary' introduction to software engineering, e.g., based on established textbooks (such as [1], [23], [24]) and apply the following modifications:

- Have one or two new lectures on formal methods in software engineering as such, that is, lectures that fully cover Learning Objectives (O1) to (O3) *once* using an as-simple-as-possible (but not trivial) formalism.
- Select some representative software description languages that are discussed in the established textbooks in an informal way and complement this discussion with a fully formal coverage, in particular of some formal semantics of these languages. Thereby, gradually complete the coverage of Learning Objectives (O1) to (O3), which in particular demand an overview over different formalisms. To fit into a given time-frame, provide formal semantics for *small and proper sub-languages* of the considered software description languages in exchange for a lengthy informal coverage of intuitive meanings.

We proceed as follows. Section III-A identifies assumptions that our course construction needs regarding, e.g., scheduling of the course and required or useful previous knowledge. Section III-B discusses how our approach relates to existing textbooks and research on software engineering. Sections III-C and III-D detail the two modifications proposed above (also cf. Table II and its discussion in Section III-D). Section III-E

TABLE II

OVERALL COURSE CONTENT AND STRUCTURE: THE ONLY NEW CONTENT BLOCK, WHICH IS DEDICATED TO THE INTRODUCTION TO FORMAL METHODS AS SUCH, IS MARKED WITH 'NEW:' (AND IN BLUE), MARKING 'COMPLEMENT: (SEMI-)FORMAL' (AND RED) INDICATES A COMPLEMENTATION OF CLASSICAL CONTENT WITH A (SEMI-)FORMAL TREATMENT.

Lecture	Topic Area	Content Overview
1	Introduction	Software, Engineering, Software Engineering; Successful software development, empirical data on project success.
2-5	Project Management	Software metrics, scales; Cost estimation (experts' and algorithmic estimation); Project, process, process modelling; Procedure models; Process models (agile, V-Model (*complement: semi-formal*)).
6-10	Requirements Engineering	Requirements, Requirements Analysis; requirements properties (completeness, consistency, etc.), kinds of requirements (tacit etc.); Dictionary; Requirements specification languages: Natural language patterns, *(new: formal methods in the software engineering context on the example of)* Decision Tables (*complement: formal*), Use Cases and Use Case Diagrams (*complement: semi-formal*), LSCs (*complement: formal*).
11-14	Design & Architecture	Model; Views; Structure Models (Class and Object Diagrams (*complement: formal*), Proto-OCL (*complement: formal*)); Design Principles and Patterns; Behaviour Models (Communicating Finite Automata (*complement: formal*), Query Language (*complement: formal*)); an outlook on UML state machines.
15-18	Quality Assurance	Test case, test execution, true and false positive/negative outcomes (*complement: formal*); Limits of testing; Glass-box testing (statement, branch, term coverage); Model-based testing and runtime-verification; Program verification (*complement: formal*), Code review.

collects some overall themes and narratives that we consider useful for courses following our proposal.

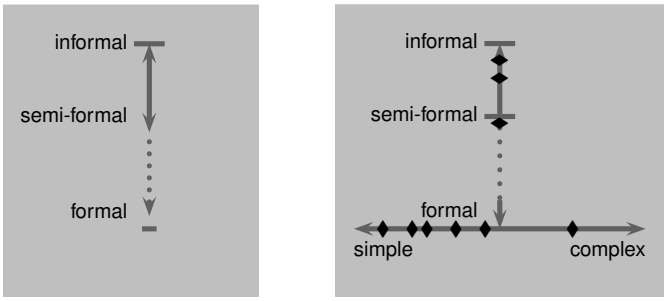
A. Pre-Requisites

The course construction described below has a few pre-requisites that are directly satisfied by many computer science curricula. We assume that an introduction to software engineering with formal methods is part of an university level curriculum for a B. Sc. in computer science (or closely related study programs). We assume that the course is scheduled later in the curriculum, e.g., the 4-th semester of a 6 semester (3 year) curriculum. More particularly, we want to assume that students have taken programming courses, introductions to algorithms and data structures, courses on networks and operating systems, databases, and (most importantly) mathematics and theoretical computer science as a pre-requisite.

An introduction to software engineering can benefit from these assumptions in the following regards. Firstly, the software engineering course does not need not introduce many concepts from mathematics or theoretical computer science. Secondly, the course can provide the narrative to see the course as using everything learned so far if it can contribute to successful software development projects (including mathematics and theoretical computer science). Thirdly, the course can draw realistic examples from the courses in applied computer science such as networks and operating systems.

B. (Lack of) Textbook Support

Unfortunately there is only little support for our teaching objectives by textbooks. The classical textbooks on software engineering (e.g., [23], [24], [26], to name a few) tend to give little room to formal methods as such, their relation to the overall software engineering context, and they usually follow what we would call an *extrapolative* approach. We illustrate our perception in Figure 2(a) as follows. On a (subjective) scale of formality (informal: free natural language text, semi-formal: precise syntax, no formal semantics; formal), classical textbooks usually only cover the scale up to semi-formal software description languages. Examples of semi-formal descriptions may obtain (often lengthy) descriptions of their



(a) Extrapolative. (b) Interpolative.

Fig. 2. Coverage of scales of formality.

intuitive meaning. Yet as discussed in Section II, a discussion of only the intuition of an apparently formal description does in general not enable students to meaningfully solve exercise or exam tasks on this formalism (cf. [25]).

Our approach is *interpolative* (cf. Figure 2(b)): We propose to discuss informal and semi-formal approaches as usual and then jump to the far end of the scale by introducing examples of formal approaches. Thereby we provide a completion of the picture as drawn by classical textbooks with "the formal end of the story". Orthogonal to the formality scale, we see a scale of simplicity (or: size of definition, expressive power, etc.) that we aim to cover to a certain amount to give an impression of the wide variety of formal software description languages. This approach sounds like adding new content and hence needing extra time that may be hard to allocate for in an existing course. To this end, we exploit the observation stated above: Classical textbooks often allocate a large amount of time on "reading out" examples of software descriptions. We propose to replace lecture blocks that consist of an overview over a software description language (often covering a lot of details) and an informal explanation, by a fully formal introduction of a smaller but proper *sub-language* of, e.g., Class Diagrams. We propose to provide a precise definition of abstract syntax and semantics, then "reading out (more complex) examples" can be delegated to the homework and tutorial sessions: Students

can use the semantics as defined in the lecture to develop a meaningful explanation of a given artefact written in the considered (and now formal) software description language.

The textbook series [27]–[29] by Bjørner takes a different approach that we would classify as inverse extrapolative: This textbook series aims at introducing software engineering completely formal with an outlook on today's informal practice. This is, in our opinion, too far away from today's software engineering practice and the background of our students. One of our objectives is to provide students with the capability to connect to today's practice based on solid vocabulary and an overview over the scales. The textbook [30] by Mills also has a strong emphasis on formality and is closer to today's practice than [27]–[29]. Yet we see as a serious drawback in [30] (and even [27]–[29]) that we perceive the connection between, e.g., Class Diagrams, and their respective formal basis too informal and example-based. In our opinion, this explicit connection is crucial to meaningfully connect course content to the background of the students.

Compared to the research literature, the course design presented here neither yields a formal methods course in a narrower sense (like, e.g., [18], [31]–[33], nor in a broader sense where outlooks on software engineering are included by means of manageable examples as in [34], [35]). The course design here in particular has, as an introduction to software engineering, a broader scope than, e.g., Noble et al. [36], [37], who focus on software modelling (as in our topic area Design, see below), Aceto et al. [38], who focus on concurrency and [39], who focus on finite-state model-checking (both included in our topic area Design).

C. Formal Methods as Such

We propose to dedicate one lecture to a comprehensive introduction to the concept of formal methods in software engineering *once* and refer back to this introduction later. The purpose of this lecture is to discuss the relations between formal methods and the software engineering context as shown in Figure 1, i.e. reading, writing, analysing, and validating formal specifications, as well as interpreting analysis outcomes. To this end, we propose to introduce one as-simple-as-possible (but not trivial) formalism and use it as a concrete example for a first discussion of disadvantages and advantages of formal methods in software engineering.

In our opinion, the topic area Requirements Engineering is a good place to host this one lecture on formal methods as such. Following the textbook by Rupp et al. [40], the topic area can begin with an introduction of vocabulary (requirement, requirement specification and analysis, etc.). Following [24], [40], desirable properties of good requirements specifications can be introduced informally, such as correctness, completeness, relevance, consistency (or absence of contradictions), neutrality, traceability, objectiveness (or testability), and being understandable, precise, maintainable, and easy to use. The established textbooks usually also discuss the opposites of these properties by pointing out the issues that can arise in a software development project if, e.g., people misunderstand each other. It is, in our opinion, useful to point out that the most prominent goal of requirements engineering is to "differentiate a correct from an incorrect system" [41], in our words: To distinguish *acceptable* system ('client has to pay') from an *unacceptable* system ('developer has to rework').

1) Informal vs. Formal: The textbooks [24], [40] indicate that today's requirements engineering is often fully based on free natural language, hence the top end of the formality scale (cf. Figure 2(b)). *Natural language patterns* [40] have been proposed to reduce the risks of misunderstandings and of leaving out important information. These patterns can be considered slightly less informal than free natural language, yet the wording used within the patterns may still be subject to different interpretations.

We can then announce to the students that we jump to the far lower end of our formality scale (cf. Figure 2(b)). We propose to begin with our definition of formal methods (cf. Section I). To make the definition come to live, we propose to use one as-simple-as-possible (but not trivial) example that satisfies the definition. To focus the teaching on concepts, the presentation of the formalism as such should (at best) not distract from this goal. There has to be a formal syntax and semantics, there have to be precisely defined properties, and there have to be techniques and tools to analyse artefacts for properties. Section IV gives an exemplary implementation of this block using Decision Tables [23], [24], [42] as a formalism.

2) Formal Description vs. System: To relate a formal description to a product ((4) in Figure 1), we usually have the formal description written in terms of so-called *observables*, i.e., formal representations of situations that are perceptible in the real world and, e.g., have a binary classification into present and absent. With Decision Tables, conditions and actions are observables, with Statecharts, events are observables, as well as states and variables. Observables may model conditions of the system environment, the (software) system as a black box, as well as internal state of the considered (software) system.

In the simplest case, an observation of a situation of a real-world system, that reacts by certain actions to stimuli, can be formally represented as a valuation σ of the observables. If the set of observables *Obs* only includes boolean observables, then the valuation is a mapping $\sigma : Obs \to \{true, false\}$. We then say that a system observation is *allowed* by a formal specification if and only if the representation of this observation as a valuation σ satisfies the specification in a well-defined sense. For Decision Tables, for example, the satisfaction relation is defined using propositional logic formulae that are mechanically derived from the rules in the table. If a system shows a non-allowed behaviour, it does not correctly realise the specified behaviour and should thus not be accepted.

By using formal methods in this way, we immediately obtain objectiveness and precision as demanded by the textbooks (cf. (5) in Figure 1): To decide whether a particular system behaviour is acceptable, we abstract the observed (or imagined) system behaviour into a valuation and mechanically check whether it satisfies the specification (using the formal semantics of the software description formalism).

3) Interpreting Formal Analysis Outcomes: Some informal properties from requirements engineering can be formalised for certain software description languages. For example, for Decision Tables, the property of completeness for requirements specification corresponds to a precisely defined notion of *(formal) completeness*.

It is important to point out that formal completeness and completeness are different and (in general) independent notions: When analysing, e.g. a Decision Table for (formal) completeness, the outcome only indicates whether each possible observation enables at least one rule. The notion of completeness from [40] is different: A requirement (existing in the client's head) is complete if and only if all necessary cases are addressed.

An outcome of analysis for (formal) completeness can relate in four possible ways to completeness (cf. Figure 3). In our experience, grasping the implications of the four cases is not easy for all students. The easier to understand case seems to be (formal) incompleteness. Possible reasons are mistakes of the engineer when writing the Decision Table or a case not named by the client. In this case, the engineer can obtain some (or all) *formal* counterexamples from an analysis and should consult the client to resolve the issue together. The *true positive* case is also imaginable (everything was done right) but the *false positive* case needs explanation. Possible reasons are misunderstandings in the discussion with the client, mistakes when writing the formalisation, or flaws in the analysis procedure.

		formally complete ✘	formally complete ✔
complete	✔	*false negative*	*true positive*
complete	✘	*true negative*	*false positive*

Fig. 3. Interpretation of formal analysis results.

With this discussion, we point out that an analysis result like 'Decision Table is complete' ('code is correct', etc.) only implies that, if the formalisation of the requirement (the design, the program, etc.) is valid (see below) and if the analysis was conducted with a flawless tool, etc., then the system will not fail *due to* an error in this requirement (this design, this code, etc.) that is discoverable by this analysis. The aspect of interpreting outcomes of formal analyses can be assumed to be crystal clear to the formal methods community (cf. [32] or dependability cases [43]), yet is (in our perception) not often stated this explicitly when discussing the application of formal methods in software engineering.

4) Formalisation as a Creative Act: Regarding the creation of formal software descriptions by engineers, the in our opinion most important aspect to point out is that observables are chosen by the engineers and choosing appropriate observables is a *creative act*. Engineers may err in two directions: Firstly, what is not represented in the 'formal world' (by observables) cannot be formally constrained and analysed for. Hence, some issues with the considered requirements may be overlooked if not represented properly. Secondly, considering irrelevant observables may make a formal specification unnecessarily hard to read or analyse. Yet we do not suggest to not have

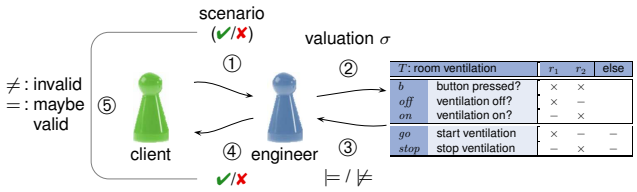


Fig. 4. Validation of formal specifications with clients.

tasks on writing formal specifications at all. We propose to have a few exercises on the creation of formal specifications, yet limited to the simpler formalisms. We consider the writing of sophisticated formal specifications beyond our learning objectives and leave this to dedicated, full semester formal methods courses.

5) Validating Formal Software Descriptions: An important property of a formal specification is to be *valid*, i.e., to correctly formalise the client's needs (in the case of requirements engineering). In the course, we point out that a formal specification written by an engineer only expresses the understanding of the engineer (modulo mistakes made when writing the specification). In general, it needs to be checked that a formal specification is valid.

We propose to touch upon a misconception that we in particular observe in conversations with practitioners: Formal methods are, in our opinion and today, first of all a tool *for the engineers*. To get *the engineer's* thoughts clear, to avoid *the engineer* misunderstanding existing write-ups, for *the engineer* to communicate with software people from the disciplines of design, quality assurance, etc. Still, the client side is in general needed for validation. We propose to use the following analogy from law to address Relation (2) in Figure 1: If a person who has not studied law needs a complicated individual contract, he or she will use the service of a lawyer. Our (non-lawyer) person can not check a contract proposal for suitability because the contract will be written in 'law language'. The contract needs to be validated with the help of the lawyer by *testing* certain scenarios: If the other party does this, what does the contract imply? Then the lawyer interprets the contract, states the consequences, and if they do not meet the expectations of the client, the contract needs to be reworked. We suggest the same for formal specifications (and illustrate the procedure by Figure 4): If the client gives a scenario (with expected outcome, ①), then *any* engineer trained in the formalism in question can (in case of Decision Tables) represent the scenario as a valuation and (objectively and reproducibly) determine whether this scenario is accepted (②, ③), can communicate the outcome to the client, and compare outcome and expectation (④, ⑤).

6) The Economics of Formal Methods: The dependability discussion in Section III-C3 leads to the aspect of economics of using formal methods: The effort for formalisation mitigates the risk of failures. Whether this effort is economic depends on the product and the project. As a first guideline towards economic use of formal methods, we suggest to focus on the

most critical system aspects and components and aim at a manageable formalisation that reflects the critical properties (cf. [20]–[22] and, e.g., [44]).

D. Complementation of Classical Content

Table II outlines a course design for a one semester course comprising 18 lectures à 90 minutes, but our approach as well extends to two-semester courses. We consider a structuring into the four topic areas Project Management, Requirements Engineering, Design, and Quality Assurance. The selection of (informal) content follows the textbooks [24], [40] and considers recommendations from the Guide to the Software Engineering Body of Knowledge (SWEBOK) [42].

We observe that a classical content selection as the one shown in Table II provides many topics where the introduction of formal methods (cf. Section III-C) can connect to. For example, topic area Project Management includes process models that can be equipped with a precise concrete syntax (thereby introducing a semi-formal description language). The content for topic area Requirements Engineering usually includes an outlook to different specification languages, including the as-simple-as-possible (but not trivial) specification language from Section III-C (bottom-leftmost black lozenge in Figure 2(b)). Scenario description languages such as sequence diagrams can serve to emphasise the difference between concrete and abstract syntax, and can provide an example of a complex formal software description language (bottom-rightmost black lozenge in Figure 2(b)). By placing the comprehensive introduction of formal methods as such following Section III-C as second or third lecture of topic area Requirements Engineering, we obtain a good progression of difficulty, ranging over formalisms with more involved abstract syntax such as sequence diagrams to more complex formalisms such as Statecharts and program verification.

Topic area Design typically covers UML Class and Object Diagrams [45], and possibly the Object Constraint Language (OCL) [46], for structural software modelling, and variants of Harel's Statecharts [9], and possibly some temporal logic, for behavioural software modelling. Selecting proper sub-languages (rather than discussing every bit and detail) enables us to present these formalisms fully formally, that is, with a proper formal semantics.

Each topic area can begin with the usual introduction of vocabulary [47], [48], and report informal practices (the topmost black lozenges in Figure 2(b)) and issues identified in the literature. To make the course's points clear(er), we suggest to follow Brakman et al. [49] and Aceto et al. [38] who ask to discuss formalisms 'in their natural habitat' with real-world examples. We expect that particularly challenging real-world settings, such as legally independent client and developer (who need to agree on a development contract including requirements specifications) or safety-critical systems for design analysis enable students to sense (or interpolate) the scale to the less critical systems. Consequently, we do not ask for one example that is considered throughout the course for each formalism. We consider the risk too high that the artificial nature of a made-up example raises more confusion than clarity if we are not able to provide satisfying answers to intricate questions about the example. To make examples manageable, we rely on the power of abstraction. For example, the operating system course provides the problem of mutual exclusion and there are concise models that provide the essence of the mutual exclusion problem.

In addition, we recommend to include ideas from well over 20 years of research on teaching formal methods, such as using (abstract essences of) real-world examples as far as possible [49], uniformly discuss formalisms [31], i.e., emphasise the re-appearing aspect of syntax and semantics (where we add the explicit distinction between concrete and abstract syntax), and choose representatives [31] (where we add the proposal to explicate *what* we want to represent).

Note that our approach does not classify as what Davies et al. [18] call 'subterfuge' and we do not aim at "use formal methods without [...] even realising it" [50]. One of our objectives is that students understand what a formal method *is*, which problems formal methods are able to solve, and which advantages and disadvantages there are. To this end, we propose to clearly communicate during the course where on the scale we are currently located.

E. Themes and Narratives

We suggest to employ the following themes and narratives throughout an introduction to software engineering complemented with formal methods. Next to emphasising the engineering aspect (cf. Section I), teaching should be based on evidence and standard documents (we do not want to 'sell' content (cf. Section V)). Another theme is to acknowledge that software engineering is to a large extent about humans in any imaginable role. We propose to stress this theme in the very first lecture and point out that many courses that students may have attended before (programming, networks, theoretical computer science, etc.) can well be taught and studied while disregarding this aspect. Software engineering can not, and we in particular understand formal methods as tools that, e.g., may reduce misunderstandings in the communication of software engineers on requirements or design ideas.

A didactic theme that we propose is to strive to 'tell the whole story', rather than presenting a selection of content without indicating the full extension. We believe that learning benefits from seeing that the presented material has defined boundaries and can be fully mastered. Therefore, we consider, for example, a small, proper sub-language of OCL [46] (with an, in our opinion, more palatable concrete syntax) whose definition fits onto two slides. Our focus is on *concepts* of, e.g., OCL rather than on coverage of all bits and details.

IV. Exemplary Course Implementation

In this section, we outline a concrete implementation of the course design presented in Section III. Regarding content and schedule, our course implementation directly follows Table II. From the existence of our implementation, we can conclude that it is possible to combine classical software engineering

TABLE III
A FORMAL SEMANTICS FOR DECISION TABLES.

A *Decision Table* T *over* C (*conditions*) *and* A (*actions*) with $C \cap A = \emptyset$ is a labelled $(m+k) \times n$ matrix

T: decision table		r_1	$\cdots$	r_n
c_1	descr. of cond. c_1	$v_{1,1}$	$\cdots$	$v_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_m	descr. of cond. c_m	$v_{m,1}$	$\cdots$	$v_{m,n}$
a_1	descr. of action a_1	$w_{1,1}$	$\cdots$	$w_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_k	descr. of action a_k	$w_{k,1}$	$\cdots$	$w_{k,n}$

where $c_1, \ldots, c_m \in C$, $a_1, \ldots, a_k \in A$, $v_{1,1}, \ldots, v_{m,n} \in \{-, \times, *\}$, and $w_{1,1}, \ldots, w_{k,n} \in \{-, \times\}$.

Columns $(v_{1,i}, \ldots, v_{m,i}, w_{1,i}, \ldots, w_{k,i})$, $1 \leq i \leq n$, are called *rules*, with *rule name* r_i, *premise* $(v_{1,i}, \ldots, v_{m,i})$, and *effect* $(w_{1,i}, \ldots, w_{k,i})$.
Each rule r of table T is assigned to a *propositional logical formula* $\mathcal{F}(r)$ over atoms $C \cup A$ as follows:

$$\mathcal{F}(r) := \underbrace{\bigwedge_{1 \leq i \leq m} F(v_i, c_i)}_{=:\mathcal{F}_{pre}(r)} \wedge \underbrace{\bigwedge_{1 \leq j \leq n} F(w_j, a_j)}_{=:\mathcal{F}_{eff}(r)}$$

where $F = \{(\times, x) \mapsto x, (-, x) \mapsto \neg x, (*, x) \mapsto \mathit{true}\}$.
Decision table T is called *(formally) complete* if and only if the disjunction of the premise formulae is valid, i.e. if $\models \bigvee_{r \in T} \mathcal{F}_{pre}(r)$,

content with the cross-cutting aspect of formal methods in a one-semester time-frame (cf. Research Question (RQ2)).

For the as-simple-as-possible formalism we have chosen *Decision Tables*. Decision tables occur in popular software engineering textbooks, e.g., [23], [24] and have a strong correspondence to business rules [51]. That the formalism of Decision Tables is non-trivial becomes tangible with exercises where the size of a table is beyond 5 by 5 (and a whole A4 page of natural language text is formalised). We have equipped Decision Tables with a logical semantics where each rule is assigned a propositional logic formula (cf. Table III). Simplicity is indicated by the fact that the definition of syntax and semantics fits onto a fraction of an IEEE page in form of Table III. In addition, propositional logic qualifies as a simple semantical domain. For details of the (semi-)formal software description languages that we discuss in our implementation of the course, including Decision Tables and almost the full language of Live Sequence Charts [52], [53], we refer the reader to [14].

In the topic area Design & Architecture, we formally introduce sub-languages of Class and Object Diagrams [45] and OCL [46] for modelling software structure. This choice is motivated by the popularity of these (semi-)formalisms in the classical textbooks and in today's software engineering practice. To model software behaviour, we formally introduce Communicating Finite Automata (CFA), that are basically Uppaal's Timed Automata [54] without time, and a fragment of the Computation Tree Logic (CTL). CFA have an appealingly concise definition of their semantics and are, in our opinion, sufficiently close to the popular Statecharts [9] for our purposes. With respect to tool use, we follow [55] (value concepts over tools) rather than a strict interpretation of [31] (tools teach methods). For behavioural modelling and software model-checking, we feel tools essential for the presentation and the exercises. For CFA modelling and analysis, we strongly advocate for the (lesser used [56]) tool Uppaal [38], [54] due to the unmatched simplicity of its user interface. For details of the (semi-)formal description languages that we discuss in our implementation of the course, we refer the reader to [15].

In the topic area Quality Assurance, we introduce program verification in the sense of Hoare [57] for sequential while programs following [58] and use the Verifying C Compiler (VCC) [59] as tool to show that program verification is possible in theory and in practice.

Following the discussion in Section III-D, we do not develop one single example that each formalism is applied to. Additionally, although we present formalisms with their 'natural' semantical domain, we address the students' demand to see how everything fits together by a unifying, abstract, formal definition of software. For our course, we define software as a finite description of a (possibly infinite) set of (finite or infinite) *computation paths*. We leave out the topic of refinement and the *formal* relation of different semantical domains. With CFAs we have one example of a working integration of a structure model (variables and events) with a constructive (the Statecharts) and a reflective (CTL formulae) formal description of behaviour. This example has been sufficient to convince the students that formalisms with a useful common semantical domains exist.

TABLE IV
PARTICIPANTS' FIELD OF STUDY AND PURSUED DEGREE.

	B. Sc. (CS, ESE)		M. Sc. (CS, ESE)		Other
2016 (131)	51.9 %	0.8 %	21.4 %	18.3 %	7.6 %
2017 (140)	64.3 %	5.7 %	5.0 %	18.6 %	6.4 %
2018 (117)	68.4 %	1.7 %	9.4 %	11.1 %	9.4 %
2019 (108)	70.4 %	2.8 %	9.3 %	9.3 %	8.2 %

V. COURSE EVALUATION AND EXPERIENCE

In this section, we report on experience from teaching the course implementation presented in Section IV.[2] We use student evaluation data to address Research Question (RQ3) of whether teaching this content is over-straining to students wrt. to workload or level.

Table IV gives the number of participants and their background. The majority are students from a B. Sc. in Computer Science (CS) study plan. A non-negligible amount of students is on the M. Sc. study plans CS or Embedded Systems Engineering (ESE), others are foreign exchange students, prospective teachers, etc. In the first week of each season, we also ask the students to subjectively self-assess their previous

[2]The material of the 2019 season is accessible at the course's homepage at https://swt.informatik.uni-freiburg.de/teaching/SS2019/swtvl

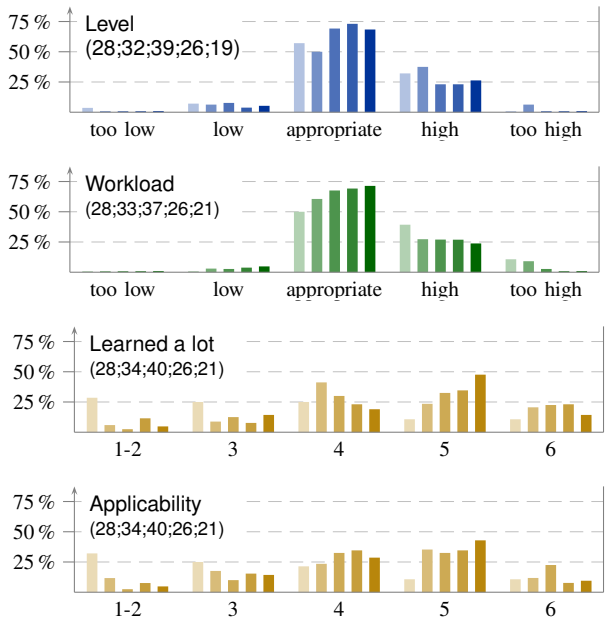


Fig. 5. Course evaluation results (2015 (light) to 2019 (dark); number of responses in parentheses, cf. Table IV for total number of participants).

experience in the four topic areas on a scale from 0 (none) to 10 (worked in commercial context). Over the seasons, the minimum has been 0 for all topic areas, the median (!) was at 1 or 2, and the maxima range from 7 to 10.

The course is regularly subject to a standard evaluation by the students. Since the evaluation is purely subjective, we consider two aspects most relevant (cf. [60]): The perceived level of difficulty and the workload relative to credit points (see the two top-most graphs in Figure 5). The results are exactly the outcome that we aim for: We want to offer a course that is challenging but not over-straining. The two bottom-most graphs in Figure 5 give the figures of two other interesting questions: The majority of responses agrees (values 4–6) to the statement 'I have learned a lot in this course' and a majority agrees to the statement 'I can (sic!) apply my acquired knowledge to different tasks'. Taking this question literally, we read it as a strong (and possibly unexpected) statement for some success regarding our learning objectives.

Truly measuring whether we have reached Learning Objectives (O1) to (O3) is notoriously difficult, as we would need to see the students in an industrial working situation. As an approximation, we can take the artificial situation of the exam. The exam to our course covers the majority of the formally introduced software description languages. Since we comprehensively introduce the formalisms of the course, we can provide completely new tasks each season inside the scope of the course's presentation. With changing exam tasks, we observe quite stable exam results over the seasons (tasks meant to be easy see high scores with low variance, more discriminating tasks see a higher variance) and take this as an indication of not completelely missing our teaching goals.

In addition to the figures, we see three notable observations in the free-text feedback of the evaluation and personal communication. Firstly, we do not have any indications of the (often reported) 'mathphobia' (e.g. [61]) but rather the opposite: Students appreciate that the course is not completely informal but provides solid 'wrongs and rights' on, e.g., syntax and semantics questions (yet of course not for the more interesting questions whether a given model is a good model, cf. [62]). A reason may be that we use mathematics on a strict as-needed basis and that we share the view "Formal models of computing systems can be developed using very expressive and flexible, but mathematically rather simple, formalisms" of Aceto et al. [38]. Secondly, we observe little difficulties with motivating our course (e.g. [63]) but quite the opposite: Compared to the first season, we were even able to reduce the effort for generic motivational words. Focusing on the technical motivations for the course content, the students seem to see the points as naturally as presented in [22]. Thirdly, we chose to neither assume a defensive position, that is, to argue against myths [64] nor an attitude of 'selling' something. In our perception, it feels challenging in first place to *convey* these myths which we then should argue against. People from the software industry did not at all object against our approach during presentations and follow-up discussions.

A re-occuring question in discussions with colleagues is whether the success of the course design presented here is bound to the particular lecturer. Unfortunately, we do not yet have experience with transferring the coure to different lecturers. Yet we do have the strong feeling that, if supported by a proper script or textbook, our course design should transfer to many different contexts. Firstly, the course has been created (or 'engineered' [25]) from the start with a clear understanding of learning objectives and with a strong constructive alignment [65], hence we consider the success of the course explainable and reproducible. Secondly, we have clearly defined pre-requisites (that are satisfied by many German and European study plans for computer science) whose suffiency has been tested on heterogeneous audiences.

VI. Conclusion

We have presented learning objectives and a generic approach (or design) to comprehensively complement undergraduate introductions to software engineering with formal methods. Experience from an implementation of this design indicates that it is possible to teach concepts of formal methods and a broad range of formalisms without sacrificing the connection to today's informal practices in software development. Evaluation results do not show any indications of over-straining the students with our approach wrt. level or workload.

References

[1] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2016.
[2] D. L. Parnas, "Really rethinking 'formal methods'," *IEEE Computer*, vol. 43, no. 1, pp. 28–34, 2010.
[3] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Oriel College, Oxford, 1998.
[4] J. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[5] B. Dongol, L. Petre, and G. Smith, Eds., *FMTea, Proceedings*, ser. LNCS, vol. 11758. Springer, 2019.

[6] J. P. Bowen, R. W. Butler, D. L. Dill, R. L. Glass, D. Gries, A. Hall, M. G. Hinchey, C. M. Holloway, D. Jackson, C. B. Jones, M. J. Lutz, D. L. Parnas, J. M. Rushby, J. M. Wing, and P. Zave, "An invitation to formal methods," *IEEE Computer*, vol. 29, no. 4, pp. 16–30, 1996.

[7] D. Bjørner and K. Havelund, "40 years of formal methods," 2014, talk.

[8] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, no. 9, pp. 8–24, 1990.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *SCP*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[10] W. Damm, B. Josko, H. Hungar, and A. Pnueli, "A compositional real-time semantics of statemate designs," in *COMPOS*, ser. LNCS, W. P. de Roever, H. Langmaack, and A. Pnueli, Eds., no. 1536. Springer, 1997.

[11] A. Moitra, K. Siu, A. W. Crapo, H. R. Chamarthi, M. Durling, M. Li, H. Yu, P. Manolios, and M. Meiners, "Towards development of complete and conflict-free requirements," in *RE*. IEEE, 2018, pp. 286–296.

[12] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post, "Scalable analysis of real-time requirements," in *RE*, D. E. Damian, A. Perini, and S. Lee, Eds. IEEE, 2019, pp. 234–244.

[13] S. Feo-Arenis, B. Westphal, D. Dietsch, M. Muñiz, A. S. Andisha, and A. Podelski, "Ready for testing: ensuring conformance to industrial standards through formal verification," *FAoC*, vol. 28, no. 3, pp. 499–527, 2016.

[14] B. Westphal, "An undergraduate requirements engineering curriculum with formal methods," in *REET@RE*, M. Moshirpour, M. Moussavi, A. M. Grubb *et al.*, Eds. IEEE Computer Society, 2018, pp. 1–10.

[15] ——, "Teaching software modelling in an undergraduate introduction to software engineering," in *EduSymp@MODELS*, L. Burgueño *et al.*, Eds. IEEE, 2019, pp. 690–699.

[16] ——, "Formale methoden in der Softwaretechnik-Vorlesung," in *SEUH*, V. Thurner, O. Radfelder, and K. Vosseberg, Eds., vol. 2358. CEUR-WS.org, 2019, pp. 21–33.

[17] L. W. Anderson *et al.*, Eds., *A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2001.

[18] J. Davies, A. Simpson, and A. P. Martin, "Teaching formal methods in context," in *TFM*, ser. LNCS, C. N. Dean and R. T. Boute, Eds., vol. 3294. Springer, 2004, pp. 185–202.

[19] M. Loomes, B. Christianson, and N. Davey, "Formal systems, not methods," in *TFM*, ser. LNCS, C. N. Dean and R. T. Boute, Eds., vol. 3294. Springer, 2004, pp. 47–64.

[20] F. L. Bauer, "Software engineering," in *IFIP Congress (1)*, 1971, pp. 530–538.

[21] C. M. Holloway, "Why engineers should consider formal methods," in *16th Digital Avionics Systems Conference, Proceedings*, vol. 1, 1997, pp. 1.3–16.

[22] L. Lamport, "Who builds a house without drawing blueprints?" *CACM*, vol. 58, no. 4, pp. 38–41, 2015.

[23] H. Balzert, *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*, 3rd ed. Spektrum, 2009.

[24] J. Ludewig and H. Lichter, *Software Engineering*, 3rd ed. dpunkt, 2013.

[25] T. Lehmann *et al.*, "Lecture engineering," in *SEUH*, A. Schmolitzky *et al.*, Eds., vol. 1332. CEUR-WS, 2015, pp. 103–109.

[26] I. Sommerville, *Software Engineering*, 9th ed. Pearson, 2010.

[27] D. Bjørner, *SWE, Vol. 1: Abstraction and Modelling*. Springer, 2006.

[28] ——, *SWE, Vol. 2: Specification of Systems and Languages*. Springer, 2006.

[29] ——, *SWE, Vol. 3: Domains, Requirements and Design*. Springer, 2006.

[30] B. Mills, *Practical Formal Software Engineering*. Cambridge U Press, 2009.

[31] A. Cerone, M. Roggenbach, B.-H. Schlingloff *et al.*, "Teaching formal methods for software engineering - ten principles," *informatica didactica*, vol. 9, 2011.

[32] K. Robinson, "Reflecting on the future: Objectives, strategies and experiences," in *FORMED*, Z. Istenes, Ed., 2008, pp. 15–24.

[33] J. P. Gibson, E. Lallet, and J.-L. Raffy, "How do i know if my design is correct?" in *FORMED*, Z. Istenes, Ed., 2008, pp. 61–70.

[34] S. Liu, K. Takahashi, T. Hayashi, and T. Nakayama, "Teaching formal methods in the context of software engineering," *SIGCSE Bulletin*, vol. 41, no. 2, pp. 17–23, 2009.

[35] S. da Rosa, "Designing algorithms in high school mathematics," in *TFM*, ser. LNCS, C. N. Dean and R. T. Boute, Eds., vol. 3294. Springer, 2004, pp. 17–31.

[36] J. Noble, D. J. Pearce, and L. Groves, "Introducing Alloy in a software modelling course," in *FORMED*, Z. Istenes, Ed., 2008, pp. 81–90.

[37] E. Sekerinski, "Teaching the mathematics of software design," in *FMED*, R. T. Boute and J. N. Oliveira, Eds., 2006, pp. 53–58.

[38] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba, "Teaching concurrency: Theory in practice," in *TFM*, ser. LNCS, J. Gibbons and J. N. Oliveira, Eds., vol. 5846. Springer, 2009, pp. 158–175.

[39] C. Jard, "Teaching distributed algorithms using spin," in *FORMED*, Z. Istenes, Ed., 2008, pp. 101–110.

[40] C. Rupp and die SOPHISTen, *Requirements-Engineering und -Management*, 6th ed. Hanser, 2014.

[41] A. Post, J. Hoenicke, and A. Podelski, "rt-inconsistency: A new property for real-time requirements," in *FASE*, ser. LNCS, D. Giannakopoulou and F. Orejas, Eds., vol. 6603. Springer, 2011, pp. 34–49.

[42] P. Bourque and R. Fairley, Eds., *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE, 2014.

[43] D. Jackson, "A direct path to dependable software," *CACM*, vol. 52, no. 4, 2009.

[44] J. E. Barnes, "Experiences in the industrial use of formal methods," *ECEASST*, vol. 46, 2011.

[45] OMG, "UML, Version 2.5.1," OMG formal/2017-12-05, 2017.

[46] ——, "OCL, version 2.4," OMG formal/2014-02-03, 2014.

[47] IEEE, *Standard Glossary of Software Eng. Terminology*, 1990, Std 610.12-1990.

[48] ISO/IEC/IEEE, *Systems and software eng. – Vocabulary*, 2010, 24765:2010(E).

[49] H. Brakman, V. Driessen, J. Kavuma, L. N. Bijvank *et al.*, "Supporting formal method teaching with real-life protocols," in *FMED*, R. T. Boute and J. N. Oliveira, Eds., 2006, pp. 59–68.

[50] J. M. Wing, "Weaving formal methods into the undergraduate computer science curriculum," in *AMAST*, ser. LNCS, T. Rus, Ed., vol. 1816. Springer, 2000, pp. 2–9.

[51] B. B. Silva, *Verification of Business Rules Programs = Verifikation von Geschäftsregel-Programmen*. Springer, 2012.

[52] W. Damm and D. Harel, "LSCs: Breathing life into Message Sequence Charts," *FMSD*, vol. 19, no. 1, pp. 45–80, Jul. 2001.

[53] J. Klose and H. Wittke, "An automata based interpretation of LSCs," in *TACAS*, ser. LNCS, T. Margaria and W. Yi, Eds., vol. 2031. Springer, 2001, pp. 512–527.

[54] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *STTT*, vol. 1, no. 1, pp. 134–152, Dec. 1997.

[55] M. Glinz, "The teacher: "concepts!" the student: "tools!"," *Softwaretechnik-Trends*, vol. 16, no. 1, 1996.

[56] S. Akayama, B. Demuth, T. C. Lethbridge *et al.*, "Tool use in software modelling education," in *MODELS*, T. C. Lethbridge *et al.*, Eds., vol. 1134. CEUR-WS, 2013.

[57] C. A. R. Hoare, "An axiomatic basis for computer programming," *CACM*, vol. 12, no. 10, pp. 576–580, 1969.

[58] K. R. Apt, F. S. de Boer, and E. Olderog, *Verification of Sequential and Concurrent Programs*, ser. Texts in Computer Science. Springer, 2009.

[59] E. Cohen *et al.*, "VCC: A practical system for verifying concurrent C," in *TPHOLs*, ser. LNCS, S. Berghofer *et al.*, Eds., vol. 5674. Springer, 2009, pp. 23–42.

[60] B. Uttl *et al.*, "Meta-analysis of faculty's teaching effectiveness: Student evaluation of teaching ratings and student learning are not related," *Stud. Educat. Eval.*, vol. 54, pp. 22 – 42, 2017.

[61] D. Mandrioli, "Advertising formal methods and organizing their teaching: Yes, but ..." in *TFM*, ser. LNCS, C. N. Dean and R. T. Boute, Eds., vol. 3294. Springer, 2004, pp. 214–224.

[62] R. F. Paige, F. A. C. Polack, D. S. Kolovos *et al.*, "Bad modelling teaching practices," in *EduSym*, B. Demuth and D. R. Stikkolorum, Eds., vol. 1346. CEUR-WS.org, 2014, pp. 1–12.

[63] J. N. Reed and J. Sinclair, "Motivating study of formal methods in the classroom," in *TFM 2004*, ser. LNCS, C. N. Dean and R. T. Boute, Eds., vol. 3294. Springer, 2004, pp. 32–46.

[64] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, no. 5, pp. 11–19, 1990.

[65] J. Biggs and C. Tang, *Teaching for Quality Learning at University*, 4th ed. Open U Press, 2011.

[66] Z. Istenes, Ed., *Formal Methods in Computer Science Education, FORMED2008, Proceedings*, 2008.

[67] R. T. Boute and J. N. Oliveira, Eds., *FMED, Proceedings*, 2006.

[68] C. N. Dean and R. T. Boute, Eds., *TFM, Proceedings*, ser. LNCS, vol. 3294. Springer, 2004.