An Undergraduate Requirements Engineering Curriculum with Formal Methods

Bernd Westphal

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

Abstract—In this article, we present a requirements engineering curriculum in the context of an undergraduate course on software engineering for students of computer science. The novelty of our approach is a broadened scope. We complement the discussion of established notions and concepts with the concept of formal methods, i.e., specification notations with precisely defined, formal syntax and semantics. To this end, we have chosen one very simple formal notation, the well-known decision tables, which allows us to give a comprehensive definition of syntax and semantics, and precise notions of completeness, consistency, vacuity, etc. in one lecture. The didactical idea is that if students are exposed to one comprehensive example, it should be significantly easier to adapt to one of the numerous formal specification notations in use (or about to be in use) in their future workplace. To complete the picture of the spectrum of formal notations, the curriculum includes Live Sequence Charts as a more expressive formal notation (which comes at the price of a more complex definition), again with comprehensive definition of syntax and semantics. The effectiveness of our curriculum is evaluated on empirical data from three seasons of teaching this curriculum.

Keywords—Requirements Engineering, Formal Methods, Curriculum.

I. INTRODUCTION

While there are textbooks dedicated to requirements engineering [1]–[4] (to name a few) as well as the corresponding sections in standard textbooks on software engineering [5]–[7], it is challenging to design an adequate curriculum for requirements engineering in an undergraduate course on software engineering. The challenge is twofold: (a) extensive volume of learning material: Dedicated textbooks on requirements engineering could support a whole one-semester course (or even a whole master’s program) on the topic, so a careful selection is necessary to give students a good basis for their further studies and their professional career. (b) increasing demand for formal methods: In our discussion with practitioners from, e.g., the automotive domain or major suppliers for the automotive branch, a recurring topic is formalisation; we perceive an increasing practical interest and use (!) of formal methods, that is, of requirements specification languages which have a precisely defined, formal syntax and semantics. There is a need for more precise requirements specification languages to mitigate the inherent risks of textual specifications (the risk of misunderstandings, to name just one) and to apply tools to analyse requirements for, e.g., consistency or completeness (see, e.g., [8]). This need even extends to small to medium sized enterprises (SME) in the branch of safety critical systems [9].

Interestingly, at present, Challenge (b) is not a special case of Challenge (a): We perceive a lack of textbooks on the general field of requirements engineering which also comprehensively cover formal methods. The textbook [1], for example, gives an excellent overview over the difficulties and (psychological) side-conditions in the field of requirements engineering as well as a broad overview over practices for informal requirements specification, yet the discussion of formal methods is limited to well-explained examples; the syntax and, most importantly, the formal semantics of formalisms are not comprehensibly presented. An example for the contrary approach is [10]–[12]. The textbook puts formal methods first, at the price of an only limited connection to today’s prevalent, informal practices of requirement specification.

The goals of the curriculum presented here as an attempt to address the challenges named above are the following:

- To give the students a high-level introduction into the most important vocabulary and concepts of requirements engineering, including awareness for existing standard documents such as [13]–[15].

- To convey the importance and the inherent difficulties of the activity of requirements engineering.

- To provide students with an overview over the spectrum of today’s requirements specification practices, ranging from fully informal (“word processor documents”), over natural language patterns, to formal specification languages. Here, the particular aim is that students should master at least one (as simple as possible) formal specification language so that they understand what we mean by “formal methods”, and so that they acquire the ability to comprehensibly solve tasks in the exercises and the exam. This means that students should be enabled to prove correctness of their solutions using the definitions from the curriculum. The exercise and exam tasks include the formalisation of the own understanding of a set of given requirements as well as formally analysing a given formal specification for certain properties like completeness.

The curriculum presented here is part of a one-semester course on software engineering. The course is a compulsory
subject of a program offering a B.Sc. degree in computer science (CS) and is usually taken in the 4th semester. In addition, some students from the M.Sc. in computer science, embedded systems engineering (ESE), and, e.g., so-called polyvalent degrees (to become teachers) have this course in their study plan. Overall, we observe between 100 and 150 students enrolled. Table I gives figures on the field of study and pursued degree as reported in the voluntary course evaluation. Regarding the field of studies, we have omitted the non-CS and non-ESE fragment; the column ‘Total’ gives the number of responses on field of studies and pursued degree, respectively. The course is supposed to give an overview over the whole discipline of software engineering. We have structured the course into four topic areas: Software project management; requirements engineering; design and architecture; and quality assurance. This article discusses the topic area requirements engineering.

Note that this paper is not supposed to teach the reader requirements engineering concepts or techniques. The contribution of this paper is our novel approach of teaching classical textbook material on requirements engineering combined with comprehensive formal methods to undergraduate students and the empirical evaluation of our approach. When reporting, e.g., our phrasing of concepts our writing may switch to a didactical tone (in the sense of direct speech).

II. Topic Area Requirements Engineering

In the following, we report on the context of our curriculum (in the overall programme and wrt. students’ previous experience with the topic area), outline the construction principles of our curriculum, and present how we realise these principles in our lectures, exercises, and tutorials.

A. Course Context and Students’ Situation

Regarding previous knowledge, our curriculum assumes basic knowledge from introductory courses on mathematics, logic and automata theory, and graph theory. Scheduling the software engineering course (of which our curriculum is a part) in the 4th semester of the B.Sc. programme satisfies this assumption. This temporal position of the course is in our opinion particularly well suited since we can build on solid theoretical ground to discuss the application of computer science theory to the practice of software development.

In the very first exercise sheet of the course (to be submitted within 3 days), we ask students for a self-assessment of their level of experience in the courses’ topic areas. We propose the following (subjective) scale:

Table I

<table>
<thead>
<tr>
<th>Year</th>
<th>CS</th>
<th>ESE</th>
<th>B.Sc.</th>
<th>M.Sc.</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>96.4%</td>
<td>3.6%</td>
<td>71.4%</td>
<td>25.0%</td>
<td>3.6%</td>
<td>28/28</td>
</tr>
<tr>
<td>2016</td>
<td>75.5%</td>
<td>26.5%</td>
<td>55.5%</td>
<td>38.2%</td>
<td>5.2%</td>
<td>34/33</td>
</tr>
<tr>
<td>2017</td>
<td>75.0%</td>
<td>22.5%</td>
<td>64.1%</td>
<td>30.8%</td>
<td>5.2%</td>
<td>40/39</td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Experience</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>responses</td>
<td>26</td>
<td>15</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0: I have no experience in that activity whatsoever. I have not taken any related subjects during my studies.
1: I have only performed the activity in the context of a lecture or programming course.
2: I have performed the activity in a project with a large user base (100+ users), a large work volume (36+ person-months) or a specific commercial purpose. I have been responsible for the planning and execution of the activity in a software development project within defined resource and time constraints.

Table II shows the results from the 2018 survey on topic area requirements engineering (based on an initial broad characterisation of the four topic areas). Of 64 responses we have a minimum value of 0, a maximum of 8, and quartiles of 0, 1, and 2. The results do not vary widely over the seasons: In 2017, we had 0-(1/1/3)-10 (min-(Q1/Q2/Q3)-max) of 88 responses, and in 2016, we saw 0-(1/1/3)-9 of 77 responses. We present the figures in the second lecture of the overall course to re-ensure the students in the lower quartile that they are not alone in their limited experience, and to inform the students from the upper quartile about the fact that the course content has been chosen to serve the majority with little experience. Experienced participants are anyway invited to share their experience with the audience.

Based on the survey results and the experience in teaching the course, we work with the following hypothesis on the situation of students. Students experience little exposition to requirements engineering topics in their previous studies. Students are used to rather precisely stated tasks in exercises (which is perfectly fine with 1st or 2nd semester courses). There is little experience with the concept of stakeholders in software development processes, which is original content of the software engineering course discussed here (for simplicity, we limit the discussion to the high-level roles customer, developer, and user). In typical exercise situations, faculty assumes the role of the customer yet exercise content is seldomly negotiable; in individual projects students assume all three roles. Up to the 4th semester, there are little long term, multi-person projects which could provide experience with the need for serious requirements documentation; lab projects (hardware and software) and seminars have a typical duration of one semester and a workload as low as a course (6 ECTS). Correspondingly, students do not imagine the size that requirements documents often reach in practice (cf. Section III) and which motivate the need for systematic and structured approaches.

B. Curriculum Design

The four topic areas in the course have a common structure. Each topic area has introductory lectures where the important concepts and vocabulary of the sub-discipline of
software engineering is introduced. The subsequent lectures then present a selection of approaches to typical problems of the sub-discipline, and of informal and formal techniques.

In the topic-area requirements engineering, we strive for a particularly wide coverage of techniques in the range between informal and formal requirements specification languages, that is, requirements specification languages which have a precisely defined, formal syntax and semantics. The main idea of the curriculum presented here is to introduce the discussed formal requirements specification languages comprehensively. That is, we give a complete formal definition of the syntax and semantics of the requirements specification languages we consider. This approach allows us to state meaningful exercises and exam tasks whose solutions can be proved to be right or wrong, thus the students are able to work on the exercise tasks with a true understanding of the formalism.

Our overall goal is to provide the students with elementary practical experience at both ends of the formality scale. An underlying hypothesis of our course is that this approach (a) gives students a feeling for the range of (semi-formal) languages in the middle (we expect it to be more difficult to extrapolate from an introduction of only informal and semi-formal languages to the practice of formal languages) and (b) allows students to adapt to other formal requirements specification languages if needed.

Highly sophisticated formal specification languages (like UML 2.x state machines with formal semantics) are clearly out of scope for an introductory course like the one considered here. Our approach has the goal to teach the concept of formal methods in requirements engineering, rather than every bit and detail of a particular, highly expressive formalism. Hence we have selected one formalism which is as simple as possible and still serves the purpose of conveying the concept of formal modelling and formal requirements analysis and one more complex formalism to give an impression of the range of formal requirements specification languages. Thus next to the scale between informal and formal languages, we attempt to cover the orthogonal scale between simple and more involved approaches.

In the curriculum presented here, the choice fell on a variant of decision tables as the simplest formalism and (a simplified variant of) Live Sequence Charts [16]–[18] (LSCs) as the more involved formalism. The choice of decision tables is mainly motivated by their simplicity, popularity in textbooks [1], [6], and their close relation to the well-known concept of business rules which are used, e.g., in web-shop software to determine the discount conditions for customers. The choice of LSCs is motivated by our extensive experience with this visual formalism [19] and their close relation to (informal or semi-formal) use cases [20]. The construction of the curriculum is not limited to these two formal requirements specification languages.

The importance of precise requirements specifications is elaborated in the subsequent topic areas of design modelling and verification and quality assurance. We demonstrate, e.g., that a design model can be formally verified against precise requirements, that expected values for test cases can be computed from precise requirements, and that sequence charts can be translated into observers for run-time verification and can serve as precise specifications of acceptance tests. In particular decision tables re-appear in the exercises on testing to reinforce the notation.

Our (admittedly ambitious) overall goal is, colloquially put, that “none of the textbooks referred to in the course (and in this paper) should bear major surprises to the students regarding concepts after taking the course”, but the books should be perceived as a refined and elaborated treatment.

C. Lectures, Assignments, and Tutorials

The topic area requirements engineering is covered by the following five lectures (à 90 minutes with a 10 minutes break in the middle):

1) Lecture 1 & 2: RE concepts and vocabulary.
2) Lecture 3: Formal methods in RE.
3) Lecture 4 & 5: Use Cases and scenarios, informal and formal notations, a formal notion of software, software specification, and software satisfies specification.

The lectures are canonical slide presentations with some room for online questions and discussions. The screen and the lecturer’s speech are recorded and available for self-study and repetition. The slides are primarily meant to support the lecturer’s spoken presentation of explanations and examples, yet the slides are self-contained regarding formal definitions.

The topic area is complemented by two exercise sheets with corresponding tutorials. Exercise sheets are available with the first of the three lectures covered by the exercises, in total there are about 10 days to work on the exercises. Students work in groups of two to three persons and submit their solutions via a learning management system. We admit a bonus of 10% on the exam admission points (see below) for early submissions, that is, submissions 24h before the regular deadline which is given by the start of the corresponding tutorial session. The concept of early submissions allows us to adjust the tutorial content to particular needs of the students and to draw interesting example solutions to be presented in the tutorials. Thereby, students are supposed to practice the online assessment of other solutions to a well-known task, which they also worked on.

The exercises are carefully crafted to provide an appropriate level of imprecision and ambiguity to allow students to practice dealing with imprecision. The voluntary course evaluation at the end of the semester typically comprises one or the other complaint about imprecise tasks. In our opinion it is crucial to get used to imprecise task statements around the 4th semester although the tutors’ correction effort is significantly higher compared to precisely stated tasks (or even multiple choice tasks).
Tutorials (à 90 minutes) are held by student workers and provide time for interaction between ca. 25 students and their tutor. The purpose of our tutorials is to discuss (technical) solutions of tasks as usual and to solve further questions online based on experience from the exercise tasks. The tutors do not just present a solution but ask the audience to “develop one good solution together” and note down the result on the screen. Practical requirements engineering tasks often do not have a unique solution so the tutors assume the role of a moderator. Further questions are, for example, asking for actions a requirements engineer should take when obtaining the reported analysis result for the given decision table. In order to prepare the tutors for their session, the tutors also work on the exercises and we conduct a so-called tutor’s tutorial on the day before the students’ tutorials in the same style in which the students’ tutorials are supposed to be presented. The regular submission deadline is the start of the corresponding tutorial in order to foster interaction; we can remind the students that most of them just submitted their solution.

The students’ submissions are later examined and graded by the tutors and feedback is sent to the students. The grading follows two scales: Admission points (“good-will rating”) and exam points (“evil rating”). Points on the good-will scale are considered for exam submission and assigned based on the knowledge of the students before the tutorial. Strictly speaking ‘wrong’ solutions may achieve full admission points if the approach of the students is comprehensible and plausible from their write-up. Exam points are assigned based on the knowledge after the tutorial and in particular picky on notation issues. The exam points are supposed to give the students a feeling for the grading applied in the (artificial situation of the) exam.

III. RE Introduction (Lectures 1 & 2)

The following Section III-A gives a brief overview over our introduction to the topic area. Section III-B describes a particular exercise where we aim to provide students with some real-world experience with requirements elicitation.

A. RE Concepts and Vocabulary

Our introduction to requirements engineering is conventional and mainly follows [7] with aspects from [1]. The introductory lectures emphasise the overloaded meaning of requirement (needed condition vs. documented representation of needed condition) [14] and requirements analysis (studying user needs vs. studying requirements) [14]. Following [7], we illustrate (a) the importance of requirements documents in the development process by pointing out dependencies of further activities and (b) the fact that requirements documents may be read by many persons with different educational background. We introduce requirements on requirements (correct, complete, consistent, testable, etc.) and on specification documents (precise, understandable, maintainable, usable) [7]. These concepts are introduced informally with examples from the literature and are revisited with the formal notations for which some concepts get a precise mathematical definition. As an interactive exercise we request the audience to imagine an average software requirements specification document and ask for the number of pages. In our experience, students seem to underestimate the size of requirements specification documents; with a proper idea of the size it may be easier to see the motivation (and the sheer need) for well-organised documents.

An important notion is the classification and analysis of explicit, semi-tacit, and tacit requirements. In our experience, it seems to be hard to imagine for the students that customers “do not know what they want”. Following [7] and [1], we emphasise that requirements engineering is to a good part about the interaction between humans with different interests, different educational background, etc.

We introduce the structure of software requirements specifications following [13]. Here, we emphasise the concept of a dictionary. To this end, we present the following sentence from one of our projects [9] and ask the audience to point out the terms which need clarification (in the sense of investigating the meaning that the customer assumes behind the terms). In our experience, the reaction of the audience is firstly reserved, the sentence seems pretty clear. After pointing out one or two examples the audience usually joins in and proposes to investigate the terms set in italics below.

The loss of the system of the ability to transmit a signal from a component to the central unit is detected in less than 300 seconds and displayed at the central unit within 100 seconds thereafter.

Still, the audience usually misses some terms found in our own dictionary [9]. We view it as an advantage to discuss a dictionary from one of our projects (rather than a made-up one) since it is known to be sufficient to solve a real-world task. We are prepared to provide in-depth responses to questions and may make the full example available if students want to see “the whole story”. It seems particularly difficult for our students to change the view from “seeing meaning” (as in day-to-day life) to “seeing lack of meaning” (the requirements engineer’s view), therefore we installed the exercise described in Section III-B below.

Regarding requirements specification languages, the introductory lectures cover the informal end of the spectrum. We introduce natural language patterns [1] and defined keywords like must, should, may, etc. in RFC 2119.

B. Requirements Elicitation Exercise

One of the practical assignments of the RE introduction consist of a semi-realistic requirements elicitation task. The essential premise of this exercise is a well-understood requirement with an imprecise description. In our case, we use the convenient fact that the overall curriculum has a software lab project scheduled in the semester of our
course. The lab project’s requirements (about 10 overall) are carefully crafted to be vague, yet we are supported by an organiser of the lab project who has a clear understanding of (non-)admissible interpretations. In general, we recommend to choose an example sentence from a real-world project that has a known, consistent customer’s interpretation since it is difficult to make up a requirement that withstands scrutiny by the students without running into contradictions.

In the exercise, the task is to provide an in depth analysis of one requirement from the lab course, for example:

Game characters have to be controlled indirectly.

The exercise has three sub-tasks: (i) create a dictionary, (ii) investigate (additional) constraints on the dictionary entries, and (iii) propose a requirements specification which characterises the (non-)admissible games as well as possible. The crucial term is ‘have to’, which has quantitative and temporal aspects that do not appear in the sentence.

To save effort on our side, students interact indirectly with the customer by e-mails to the tutors. For corner-cases, questions are forwarded to the person playing the role of the customer to ensure consistent answers.

In the tutorial corresponding to this exercise, we offer the students to test their findings. We prepare a collection of (non-)admissible game examples and ask students to report whether their solution would admit the game. Here we point out that false admission is as undesirable as false rejection. The former may cause costly redevelopment, the latter may give the students a basic but direct experience with the test.

A decision table is called complete if and only if the disjunction of all rules’ premises is a tautology, i.e. if

$$\lor_{r \in T} \mathcal{F}_{\text{pre}}(r) = \top$$

(2)

For convenience, at most one else-rule can be used (see below). The semantics of the else-rule has the negation of the disjunction of all other rules’ premise formulae as premise, i.e. $$\mathcal{F}_{\text{pre}}(\text{else}) := \neg \left( \lor_{r \in T \setminus \{\text{else}\}} \mathcal{F}_{\text{pre}}(r) \right)$$. A decision table with else-rule is always complete.

Similarly, we can define useless rules. A rule $$r \in T$$ is called useless (or: redundant) if and only if there is another (different) rule $$r’ \in T$$ whose premise is implied by the one of $$r$$ and whose effect is the same as $$r$$’s, i.e. if $$\exists r’ \neq r \in T \bullet$$

$$\mathcal{F}_{\text{pre}}(r) \Rightarrow \mathcal{F}_{\text{pre}}(r’) \land (\mathcal{F}_{\text{eff}}(r) \Leftrightarrow \mathcal{F}_{\text{eff}}(r’))$$.

(3)

Rule $$r$$ is called subsumed by $$r’$$.

A decision table $$T$$ is called deterministic if and only if the premises of all rules are pairwise disjoint, i.e. if

$$\forall r_1 \neq r_2 \in T \bullet \models \neg (\mathcal{F}_{\text{pre}}(r_1) \land \mathcal{F}_{\text{pre}}(r_2))$$.

(4)

Otherwise, $$T$$ is called non-deterministic.

To introduce the concept of domain modelling [12], we extend decision tables by a conflict axiom and a simple conflict relation. A conflict axiom $$\varphi_{\text{conflict}}$$ is a propositional formula over conditions $$C$$. A decision table $$T$$ is called complete wrt. conflict axiom $$\varphi_{\text{conflict}}$$ if and only if the disjunction of all rules’ premises and the conflict axiom is a tautology, i.e. if $$\models \varphi_{\text{conflict}} \lor \bigvee_{r \in T} \mathcal{F}_{\text{pre}}(r)$$. A rule $$r \in T$$ is called
vacuous wrt. conflict axiom \( \varphi_{\mathit{conf}} \) if and only if the premise of \( r \) implies the conflict axiom, i.e. if \( | F_{\mathit{pre}}(r) | \rightarrow \varphi_{\mathit{conf}} \). A DT with conflict axiom \( \varphi_{\mathit{conf}} \Leftrightarrow \) true is equivalent to the DT with the same rules but without a conflict axiom, so the notion of conflict axioms is a conservative extension of the DT language. A conflict relation on actions \( A \) is a transitive and symmetric relation \( \subseteq (A \times A) \) on the set of actions. A rule \( r \) is called consistent with conflict relation \( \subseteq \) if and only if there are no conflicting actions in its effect, i.e. if

\[
| F_{\mathit{eff}}(r) | \rightarrow \bigwedge_{(a_1, a_2) \in \subseteq} \neg(a_1 \land a_2).
\]  

The collecting semantics of a decision table \( T \) over actions \( A \) is given by the following formula:

\[
F_{\mathit{coll}}(T) := \bigwedge_{a \in A} a \leftrightarrow \bigvee_{r \in T, r(a) = x} F_{\mathit{pre}}(r)
\]  

In the collecting semantics, \( T \) is consistent with the conflict relation (under conflict axiom \( \varphi_{\mathit{conf}} \)) if and only if there are no conflicting actions in the effect of jointly enabled transitions, i.e. if

\[
| F_{\mathit{coll}}(T) | \land \neg \varphi_{\mathit{conf}} \rightarrow \bigwedge_{(a_1, a_2) \in \subseteq} \neg(a_1 \land a_2).
\]  

### B. Lectures

In the lecture, decision tables are presented in the following order. For simplicity, we begin by introducing core decision tables without the else rule, and without conflict axiom and relation. This is sufficient to discuss the use of formal notations to precisely specify requirements. The idea is that conditions and actions are abstract models of conditions and actions which are observable for the system for which requirements are specified. The running example is a room ventilation system which is installed in the lecture hall where the lectures take place. There is a single button which switches the ventilation on or off depending on the current state of the ventilation system. We propose to model the situation of the button by DT condition \( b \). For didactical purposes, we introduce the DT conditions on and off which model the states reported by two sensors, one which measures the airflow (and reports the system to be ‘on’ if and only if there is measurable airflow) and one which measures the current supplied to the ventilators (and reports the system to be ‘off’ if and only if there is no current supplied). A real controller may just employ one sensor, yet considering two sensors is useful to discuss the concept of environment assumptions later. We consider two DT actions, go and stop, which model the procedures employed to start up the ventilation from an inactive state and to shut down the running room ventilation.

We propose the following DT \( T_1 \) to formalise the requirement that the button toggles the room ventilation between on and off depending on the current state of the ventilation.

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>room ventilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>button pressed?</td>
<td></td>
</tr>
<tr>
<td>ventilation off?</td>
<td></td>
</tr>
<tr>
<td>ventilation on?</td>
<td></td>
</tr>
<tr>
<td>go</td>
<td>start ventilation</td>
</tr>
<tr>
<td>stop</td>
<td>stop ventilation</td>
</tr>
</tbody>
</table>

The decision table formalises the requirement by the notion of allowed behaviour. A system satisfies the DT if and only if, for each observable behaviour of the system, there is at least one rule in the DT which allows this behaviour. Formally, an observation of the system such as ‘the button is pressed, the sensors report airflow and current, and the ventilation is shut down (but not started up)’ gives rise to a logical valuation \( \sigma : C \cup A \rightarrow \mathbb{B} \) of the DT conditions and actions, in this case \( \sigma = \{ b \mapsto \mathit{true}, \mathit{off} \mapsto \mathit{false}, \mathit{on} \mapsto \mathit{true}, \mathit{go} \mapsto \mathit{false}, \mathit{stop} \mapsto \mathit{true} \}. \) A model \( \sigma \) of an observation is allowed by a DT \( T \) if and only if there is a rule \( r \) in \( T \) such that \( \sigma \models F(r) \). In the room ventilation example, we have \( \sigma \models F(r_1) \) thus the behaviour modelled by \( \sigma \) is allowed by \( T_1 \). In this way, decision tables can be used to objectively specify desired system behaviour. In the lecture, we cast the requirements formalisation using DTs as a message to the developers: ‘Please provide a controller software such that in each situation (button pressed etc.), whatever the software does (go, stop) is allowed by \( T_1 \).’

To avoid confusion, we emphasise that DTs are not meant to specify all system or software behaviour and that DTs are usually meant to be read in a particular context. The DT for the room ventilation system, for example, (on purpose) abstracts from all kinds of details related to operating the sensors and implementing the go/stop procedures. And it assumes a cycle consisting of reading inputs, consulting the table, and executing actions before the cycle starts over.

Having introduced the concepts behind DTs as formal requirements specification language, we focus on requirements analysis. The first and simplest example is completeness which is one of the main properties generally desired for requirements documents [1]. With DT completeness as defined above, we obtain a formal model of completeness.

A very important part of the lecture is the discussion of the interpretation of the outcome of an analysis of a DT for (formal) completeness. We have to discuss the four cases of true and false positive and negative outcomes. The easier to grasp cases are, in our experience, the positive outcomes, that is, the table is found to be incomplete. Then there exists a counter-example, a valuation of the conditions and actions for which no rule is enabled, i.e., which does not satisfy any rule’s premise. This finding is a strong indication that there is an open issue with the requirements analysis (in the sense of understanding the customer’s needs) and should be discussed with the customer. A true positive may be a case forgotten by the customer, a false positive may be due to a misunderstanding or a mistake when writing the DT. For the negative case, that an DT is (formally) complete, we point out that such a finding in general does not imply
that the requirements are complete in the sense that the customer’s needs have been completely covered. Reasons are, for example, that DT conditions have been chosen too coarsely so that they cannot distinguish all cases relevant to the customer. What we can conclude from a true negative is that there are no incompletenesses in the scope of the DT left. In other words, that under the assumption that, e.g., DT conditions are appropriate models of the considered system, the requirements analysis is complete. Thus the risk of an incomplete requirements analysis (in the informal sense) can be lower when using formal methods such as DTs as compared to not using formal methods. Finally, we make students aware of the possibly worst case in practice: A false negative, that is, a DT which is only (formally) complete due to a misunderstanding or a mistake made by the author of the DT. The else-rule is presented as an example for the risks of convenience notations: An incautious addition of an else-rule may unintentionally hide incompleteness.

The notion of (formal) uselessness of DT rules serves us as an example with less obvious implications for the requirements analysis. If a DT is found to have useless rules, the customer may have overlooked that a certain case is a special case of another rule or it may be the result of a misunderstanding. If a rule is confirmed to be useless, it should be removed to improve readability, a general demand for requirements documents [1].

We introduce the conflict axiom as a simple example for formal domain modelling [12]. The intuition of the conflict axiom is that it characterises the set of observations which “cannot happen” due to relations between observables in the domain. If the room ventilation, for example, uses only one sensor to detect the state of the ventilation, then formally the values of observables on and off should not be independent, e.g., the conditions on and off both mapped to true may not happen at all. This environment assumption can be modelled by adding the conflict axiom $\phi_{conf} := (on \land \neg off) \lor (\neg on \land \neg off)$. The following decision table $T_3$ is complete with respect to the conflict axiom, that is, for all cases not excluded by the conflict axiom, at least one rule is enabled.

<table>
<thead>
<tr>
<th>$T_3$ room ventilation</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>button pressed?</td>
<td></td>
<td></td>
<td>$\times$</td>
</tr>
<tr>
<td>off</td>
<td>$\times$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ventilation on?</td>
<td></td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>go</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stop</td>
<td></td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$\neg((on \land off) \lor (\neg on \land \neg off))$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We use the notion of conflict axioms to point out the risks of domain modelling: If a conflict axiom is too weak (if it erroneously excludes cases which may happen in the considered system), an incompleteness of the gathered requirement may be overlooked and a software implemented according to the erroneous DT may thus lack behaviour for existing cases. An analysis of a DT for (formal) vacuity can indicate issues with the requirements gathering. The vacuous rule may indeed be redundant and should be removed for the sake of readability, or the vacuous rule is intended and the conflict axiom excluded a possible case and thus needs to be strengthened.

Conflicting actions are an example for the notion of consistency, that is, that requirements documents should be free of contradictions. From the reactions of our students it seems that inconsistent specifications are not perceived as a frequent issue (cf. Section II-A). We explain that inconsistencies happen in practice by pointing out that requirements analysis can span multiple days or weeks and that, over time, different requirements engineers may communicate with different customer representatives (RE practitioners will confirm that inconsistencies may be observed with the same person on a single day). The notion of conflicting actions is needed to discuss consistency since DTs in the standard, non-deterministic semantics are always consistent: With the DT syntax, a single rule cannot require an action to be taken and not taken at the same point in time. Thereby students observe that a carefully chosen notation (or natural language pattern) can already eliminate certain inconsistencies. We basically introduce the collecting semantics (where the actions of all enabled rules are expected in the observation) to make the consistency analysis a bit more challenging.

C. Exercises & Tutorials

In the exercises, we firstly practice the technical formal definitions. The tasks are to analyse given, abstract decision tables (which do not have a “story” like the room ventilation system) for completeness, non-determinism, and useless rules. Both, with and without considering the given conflict axiom in order to sense the effect of the conflict axiom.

In the tutorials, the students’ results are used to re-iterate the interpretation of formal analysis results, e.g., that we recommend to discuss the cases not covered by an incomplete DT with the customer. Furthermore, we re-iterate the technical aspects of DT analysis, that is, how to prove that a DT is complete or incomplete. A purely mechanical procedure to decide completeness is a truth table; thus, e.g., SAT-solving procedures can be used to easily implement tools which automatically check DTs for consistency. We also stress the advantage of objectivity we gain from using a formal notation: Given a DT, it is either complete or incomplete, and if there are different opinions on the result, then one party can easily convince the other of their result by a proof.

The technical exercises are complemented by a task to create a decision table from a given transcript of a customer interview. For two seasons, we used a description of the computation of postal charges for parcels depending on destination region and properties of the parcel. The transcript we made available to the students is a simplified version of a real project where a shipping cost computation should be implemented. An existing ambiguity in the natural language
description is intentionally preserved, so that we expect at least two correct solutions of the task; in addition, the text intentionally contains redundant and unnecessary information. In the current season, we use a made-up transcript which describes the computation of discounts in a fictitious book lending business which contains one intentional ambiguity as well as redundant and unnecessary information. The resulting decision tables are non-trivial: They range over 5 to 6 conditions, 6 to 7 actions, and 8 to 10 rules.

In the tutorial, the tutors re-iterate how to principally approach tasks of this kind and present interesting early submissions. We usually have one example of a syntactically non-well-formed DT and one example of a too coarse choice of conditions or actions, to point out the creative act of choosing appropriate sets of conditions and actions. In addition, there are one or two good solutions such that students can practise to read other students’ results and assess their quality online. The good solutions are furthermore used to resolve the ambiguity in the natural language description: The tutors report the customer’s intention. The good solutions are used to point out that decision tables, like most other requirements specification notations, do not apply equally well to all imaginable kinds of requirements.

We, e.g., observe that our notion of decision tables does not have a concise notation for disjunctions over DT conditions.

The transcripts come with one to three concrete examples of shipping cost computation or book lending discount, respectively. The grading of the students’ submissions penalises cases where a proposed solution yields wrong results for the given examples; thereby we stress the importance to test formalisations using examples with expected outcomes given by the customer. We close the DT discussion in the tutorials by posing the (provocative) question to the students whether, if they had the task to implement the shipping cost or discount computation and if the choice were either/or, they would choose the text or the decision table.

V. A MORE INVOLVED FORMALISM: LIVE SEQUENCE CHARTS (LECTURE 4 & 5)

In the last two lectures, we focus on the technique of discussing concrete scenarios with customers. Of the informal (or semi-formal) methods we discuss Use Cases [20] and Use Case Diagrams, emphasising the fact that Use Case Diagrams alone carry very little information and that the important information is the Use Case as such (pre-/post-conditions, normal and exceptional cases).

Sequence diagrams like Message Sequence Charts [22] (MSCs) and UML 2.x Sequence Diagrams are established (semi-)formal description languages for scenarios. In the course, we discuss Live Sequence Charts [16]–[18], one of the variants of sequence diagrams with an elaborate formal syntax and semantics (cf. Figure 1 for an example). The semantics of an LSC in the flavour of [18] is based on the construction of a variant of symbolic Büchi automata (called TBA; cf. Figure 2). Advocates of sequence diagram variants with formal semantics emphasise the combined advantage of being useful in formal verification efforts and serving as effective illustrations in discussions with customers (while the core computer scientist would maybe rather prefer the pure TBA). Our presentation is extensive to a certain amount; the presentation is simplified by skipping real-time aspects, the binding of system components to instance lines, and by leaving pre-charts to a high-level discussion.

The discussion of LSCs in the course serves the purpose to illustrate that more convenient or expressive formalisms may have a more involved technical background than, e.g., simple decision tables. Thereby, we intend to give the students a basic feeling for the range of available (visual) formalisms for requirements specification to be chosen according to concrete project needs. Furthermore, LSCs serve as the one visual formalism discussed in the overall software engineering course which is not reduced to the necessary essence (as are class diagrams and state charts). Using LSCs as example, we also introduce how to formalise the notion of a software satisfies a specification. We define software as a finite description \( S \) of a (possibly infinite) set \( \{ S \} \) of (finite or infinite) computation paths of the form \( \sigma_0 \overset{\alpha_1}{\rightarrow} \sigma_1 \overset{\alpha_2}{\rightarrow} \sigma_2 \cdots \) where \( \sigma_i \in \Sigma, i \in \mathbb{N}_0, \) is called state (or configuration), and \( \alpha_i \in A, i \in \mathbb{N}_0, \) is called action (or event). We define software specification as a finite description \( \mathcal{S} \) of a (possibly infinite) set \( \{ \mathcal{S} \} \) of softwares.
points, quartiles, maximum points, and the arithmetic average for points achieved in Tasks 2.1 and 2.2, and the sum of the two tasks per student (hence, e.g., the total minimum is not necessarily equal to the sum of the two minima above).

The figures show that, overall, more than 50% of the participants achieve 13 or 14 points of possible 15 points, and 75% of the participants more than 12 points. Thus we have a clear indication that it is possible to see a large majority of students having acquired good or very good skills in formally analysing decision tables and formalising textual requirements using decision tables. Formalising textual requirements using decision tables is in average solved better (more than 75% of the participants achieve 8 of 9 points) than the analysis task. Our hypothesis for the lower scores in the 2017 season is that we changed the task slightly to explicitly require all rules’ premise formulae (which some students may have overlooked). Season 2017 was the only one of the three seasons which was taught in German. The course alternates between English and German over the seasons; 2015 was taught in English as an exception.

Table IV gives the results for the LSC-related tasks in the previous three seasons. Task 3.1 is a multiple choice on formal definitions related to LSCs, Task 3.2 asks for parts of the edge annotations in the TBA (2015 and 2016) or a partial construction of the TBA (2017), and Task 3.3 asks for computation paths which violate a given chart, or are accepted with and without taking a legal exit. In the LSC tasks, students do not perform as well as with decision tables. This result is not unexpected since the LSC tasks are constructed to lie in the second-highest and highest learning levels (analysis and synthesis), so they are meant to distinguish the ‘very good’ from the ‘good’ students. Still, we are more than satisfied with the results given that in the very first season, we planned with the option to exclude the LSC tasks from the grading following the popular belief that formal methods are (too) hard to master.

VII. Conclusion

We have presented a curriculum for an introduction to requirements engineering concepts with special emphasis on
formal methods. The curriculum is taught as part of a one-semester course for an introduction to software engineering and is built around the idea of teaching exemplary formal notations (decision tables and Live Sequence Charts) in a simplified but comprehensive form with formally defined syntax and semantics. We measure the effectiveness of our curriculum by empirical data on the results of the students in the corresponding tasks of the final exam collected over three years. The curriculum appears effective towards our goal to best prepare students for their professional career in the presence of increasing demands for formal methods from industry, as well as in an academic career path.

“Customers and formal methods? Don’t even go there.” (free translation) is stated in [7], and this, in general, does not contradict our experience. Our 3 years experience with our curriculum clearly rejects the hypothesis that “you cannot even go there with undergraduate students”. To the students, we explain our agreement to Ludewig’s quote with the analogy of law: Without an educational background in law, we, the customers, also do not truly understand what the consequences of an individual contract are. The lawyers who develop the contract for us need to explain the meaning and “test” the contract with scenarios of which we know our desired outcome. In our opinion, the same holds for requirements engineering: Formal requirements specifications are first of all written to serve the requirements engineers, developers, testers, etc.; they are not primarily meant to be truly understood by all customers. Yet they need to be explained (orally and textually) to and validated with the customer by requirements engineers who are trained in formal methods.

In future work, we plan to further refine our curriculum based on more empirical data. For example, we would like to investigate which difficulties students face with the exercises and which kinds of tasks are didactically more effective to overcome these difficulties. We plan to collect more detailed data on the students’ performance wrt. the exercises and possibly integrate questionnaires into the exercise sheets to test some of our hypotheses on the students’ situation.

ACKNOWLEDGEMENTS

We would like to thank Sergio Feo Arenis for his invaluable support as a teaching assistant during the first two seasons. The course’s current form is hardly imaginable without Sergios contributions to discussions and elaboration of the exercises. Furthermore, we thank our student assistants for their committed contributions to exercises and tutorials.

REFERENCES