Automatic Generation of Learning Assignments for Software Engineering Formalisms

Michael Steinle
Department of Computer Science
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany
steinlem@informatik.uni-freiburg.de

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

***Abstract*—Creating learning or examination assignments is a re-occurring activity in the context of teaching. In particular new examination assignments are often desired for each season of teaching a course. In our teaching, we have observed that our creation of assignments for software engineering formalisms uses certain intuitive constraints to obtain assignments at a designated level of difficulty and, e.g., with particular properties to be analysed in the assignment.**

In this work, we present an approach where we formalise our (formerly intuitive) constraints and use constraint solving tools to automatically synthesise learning assignments that satisfy these constraints. In our approach we leverage the fact that our software engineering course teaches the majority of software description languages fully formal. That is, an artefact using such a software description language is then a mathematical object for which we can give precise constraints. We demonstrate our approach on the example of learning and examination assignments for the notion of determinism on decision table and discuss applications of our assignments synthesis procedure.

I. Introduction

Learning assignments are, following Seel [1], selected and prepared learning objects with the aim to initiate, control, and organise learning processes. Following Renkl [2], learning assignments in this sense are used to learn and practice knowledge and capabilities. Renkl observes that different learning assignments can lead to comparable acquisition of knowledge and that 'well meant but badly done' stimulations for thought and embeddings of learning assignment into teaching contexts can disturb the learning process. Learning assignments have thus to focus on the knowledge or capability to learn, and be designed for and embedded into the teaching context.

The question what makes a learning assignment a *good* learning assignment in general and how to use certain learning assignments in teaching is in our perception still actively researched in didactics. A requirement for most learning assignments will be that they are *solvable*. Another requirement is that learning assignments match the knowledge and experience of the learners. Advanced learners hardly benefit from easy learning assignments on the competence level 'remember' [3] while beginners may be frustrated by difficult assignments on the competence level 'create' [3].

In this work, we consider the problem of creating learning assignments with the purpose of practising analysis procedures (competence level 'apply') for formal software specification languages in the context of our undergraduate introduction to software engineering [4]–[6]. Creating learning assignments in this context is challenging because it is not always completely obvious that a proposed assignment is solvable. To lower the student's cognitive load, we also prefer to use learning assignments where a single artefact is supposed to, e.g., be analysed for multiple aspects. Therefore, the artefacts for our learning assignments need not only be solvable but they need to equally well support multiple learning goals. Hence, simple mutations of existing artefacts does not provide a reliable procedure to create new learning assignment.

When creating new assignments, e.g., to not have the same tasks in each seasons or for exams, we observed that we can identify different levels of difficulty *within* the same class of learning assignments. Furthermore, we gained the impression that we can precisely characterise those classes of difficulty using properties of the (formal) artefacts.

We propose to develop an approach to (a) *formalise* our understandings of didactical aspects of learning assignments on software description languages (like solvability, learning goals, and level of difficulty) and (b) use existing procedures and tools to *automatically generate* learning assignments with desired properties. In this article, we report on an investigation of the feasibility of this approach. To this end, we have developed, implemented, and evaluated a synthesis procedure for the formalism of decision tables.

The article is structured as follows. In Section II, we illustrate our approach on the example of learning assignment for basic calculus. Section III recalls decision tables and gives an example of an exam task that we would like to automatically generate. Sections IV and V describe our synthesis procedure and preliminary empirical results, Sections VI and VII discuss our results and related work, and Section VIII concludes.

II. An Analogy: Learning Assignments for Addition with Carry

In this section, we want to illustrate the proposed approach on an example of learning assignments for elementary school arithmetic, namely 'paper & pencil' addition with carry of two positive decimal numbers. As a teacher, one may want to present learning assignments with an increasing level of

$\begin{array}{r} 1234 \\ +\ 625 \\ \hline 1859 \end{array}$	$\begin{array}{r} 1234 \\ +\ 62_1 6 \\ \hline 1860 \end{array}$	$\begin{array}{r} 1234 \\ +_1\ 8_1 7_1 6 \\ \hline 2110 \end{array}$	$\begin{array}{r} x_n \cdots x_1\ x_0 \\ +_{c_{n+1}}\ y_n c_n \cdots c_2\, y_1 c_1\, y_0 \\ \hline z_{n+1}\, z_n \cdots z_1\ z_0 \end{array}$
(a)	(b)	(c)	(d)

TABLE I: Addition with carry can be simple, e.g. (a), intermediate, e.g. (b), or advanced, e.g. (c). Case (d) provides an abstract formal view on the task of adding two numbers.

difficulty to let pupils practice the routine of 'paper & pencil' addition without carry before moving to the general case, and possibly in between having some learning assignment with exactly one carry. So the teacher may have identified the following three levels of difficulty: Easy tasks where no carry is needed (cf. Table Ia), intermediate level tasks where exactly one carry is needed (cf. Table Ib), and the general case where one carry may cause a subsequent carry (cf. Table Ic).

To create new learning assignments for addition, e.g., in an e-learning tool, the teacher could firstly formalise the addition of two n-digit decimal numbers as shown in Table Id. The task of adding two numbers in decimal representation consists of two numbers x and y with their decimal representation $x_n, \ldots, x_0$ and $y_n, \ldots, y_0$. The first carry, caused by adding the least significant digits x_0 and y_0, is c_1, which has value 1 if and only if $x_0 + y_0 > 10$, and value 0 otherwise.

Viewing addition tasks as mathematical objects, the teacher could secondly precisely characterise the sets of addition tasks corresponding to the levels of difficulty described above. An easy addition task is one where all carries c_i, $1 \leq i \leq n$, have value 0. Intermediate level and general case tasks can be characterised similarly. With this formalisation of learning assignments and the level of difficulty, the teacher could finally use any integer constraint solver that supports addition and comparison to obtain tasks that are, e.g., in the class of easy tasks: We are asking for a solution to the constraint solving problem that the open symbols x_i and y_i are assigned values between 0 and 9 (decimal digits) such that the valuation satisfies the constraint that all carries remain 0. Note that the number of digits n is a parameter of the constraint solving problem with which the teacher can control, e.g., the time it takes to solve a task as an additional teaching aspect.

That is, we propose to formalise didactical intents on learning assignments that involve mathematical objects (such as addition tasks) in form of constraints and to use appropriate constraint solving techniques to automatically generate learning assignments for a given intent.

III. Decision Tables

Decision tables are a simple software engineering formalism that can, e.g., be used to formalise requirements [7], [8]. Formally, a *decision table* T is an $(m + k) \times n$ matrix with entries from the set $\{-, \times, *\}$. The top m rows correspond to *conditions* $c_1, \ldots, c_m$ from the set of conditions C, and the bottom k rows to *actions* $a_1, \ldots, a_k$ from the set of actions A, which is disjoint from C. Columns are called *rules*. The top m entries of a rule (the *premise*) $v_{1,i}, \ldots, v_{m,i}$ are elements

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

Fig. 1: Concrete syntax of decision tables.

of the set $\{-, \times, *\}$, and the bottom k entries of a rule (the *effect*) $w_{1,i}, \ldots, w_{k,i}$ are from the set $\{-, \times\}$. That is, '$*$' may not occur in an effect of a well-formed decision table.

The semantics of decision tables is given by a function $\mathcal{F}$ that assigns to each rule r in T a propositional logic formula over C and A. Given premise $(v_1, \ldots, v_m)$ and effect $(w_1, \ldots, w_k)$ of r, the semantics function is defined as

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

where $F := \{(\times, x) \mapsto x, (-, x) \mapsto \neg x, (*, x) \mapsto \mathit{true}\}$. For a discussion of the use of this semantics in requirements engineering we refer the reader to [4], [5].

We can formally define different properties of decision tables that are useful in requirements engineering such as completeness, consistency, existence of useless rules, etc.. For the purpose of this article it is sufficient to provide the definition of (non-)determinism. A decision table T is called *deterministic*, if and only if for all different rules r_1 and r_2 in T, their premise formulae are (logically) disjoint, i.e. if

$$\forall\, r_1 \neq r_2 \in T\bullet \models \neg(\mathcal{F}_{pre}(r_1) \wedge \mathcal{F}_{pre}(r_2)). \qquad (2)$$

Intuitively, a decision table is deterministic if and only if there is no valuation of the conditions in C that satisfies the premise formulae of two different rules.

To practice the analysis of decision tables for determinism, learning assignments can start on the competence level of 'apply'. In the course, we present a truth table-based approach to decide determinism for a given decision table: For each valuation of the observables in C note down which rules' premises are satisfied. The table is deterministic if and only if no valuation satisfies more than one rules' premises.

A proof of determinism needs an argument on the rules' premise formulae (e.g., the truth table), while a proof of non-determinism strictly speaking only needs one valuation and the argument that this valuation satisfies the premise formulae of two different rules. Hence the workload for a given deterministic table can be higher (construction of truth table) than for a given non-deterministic table (give a counter-example as explained above). Thus in the role of a teacher, we often want to control whether the decision table given as a learning assignment is deterministic or non-deterministic.

Figure 2 shows a simplified example of an exam task on decision tables. The Subtask (1) tests that students are able

T		r_1	r_2	r_3
c_1		×	×	−
c_2		×	−	∗
c_3		−	×	∗
a_1		×	−	−
a_2		−	×	−

1) Give the rule formulae for r_1, r_2, r_3.
2) It is claimed that T is deterministic. Prove this claim.
3) Does T have a useless rule? Prove your claim.

Fig. 2: Example exercise task on decision tables.

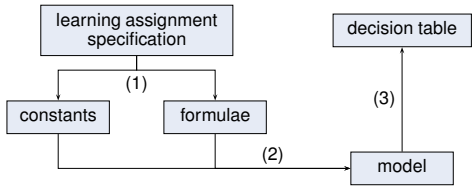


Fig. 3: Intermediate artefacts of decision table task generation.

to use the semantics function (competence level 'apply'). This kind of task is obviously solvable for each well-formed decision table, yet to be most useful, one may wish to provide a table that comprises all possible entries. Subtask (2) tests competences in applying a particular proof strategy for a particular decision table property. This subtask is only solvable (or at least not strongly confusing) if the considered decision table *has* the claimed property. Subtask (3) tests competences on analysing a decision table for whether it has a certain property such as useless rules (hence touching the competence level 'analyse'). Depending on the correct answer, different proof strategies apply that may need a significantly different amount of time to solve. One may aim at a decision table for which Subtasks (2) and (3) need different proof strategies so to test both and to limit the necessary effort. Note that using only one decision table lowers the cognitive load for the learners since only one artefact needs to be read and the learners can concentrate on the analysis tasks as such.

Creating learning assignments as shown in Figure 2 is difficult because multiple constraints need to be considered together. A change in one cell of the decision table may unintentionally render the table non-deterministic, or make the analysis for determinism too easy (for example by having two identical rules). To simplify the process of creating learning assignments or exam tasks on decision tables, we propose to exploit the fact that decision tables are introduced as mathematical objects in our course. Properties of decision tables, such as determinism, are formally defined and thus can be used as constraints in a synthesis procedure. In addition, we propose to formalise our didactical understanding of classes of learning assignments on decision tables. For example, that an analysis for non-determinism is trivial if there are two identical rules in the table. Other examples aim at the corner cases of the definition of determinism from the course and we may want to leave out these cases in beginner's exercises. The task to create, e.g., a complete decision table of a particular size then reduces to a mathematical constraint solving problem.

IV. Generating Learning Assignments on Deterministic Decision Tables

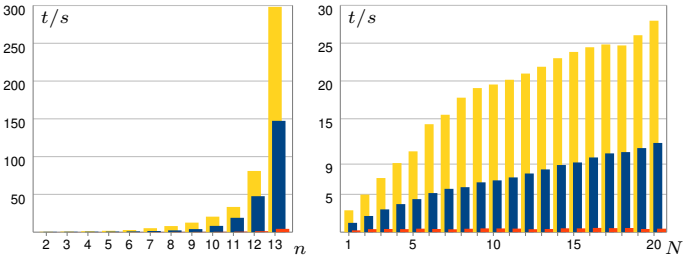
In the following, we outline a procedure to generate deterministic decision tables of a specified size. We focus on deterministic decision tables. Non-deterministic decision tables are the dual case (we negate one of the learning assignment creation constraints), yet to be useful learning assignments, non-deterministic decision tables need further didactic constraints, e.g., on the level of difficulty. We discuss some of these further constraints at the end of this section. The basic idea of our procedure to generate deterministic decision tables is to reduce the generation problem to an SMT-problem. SAT modulo theory (SMT) is a natural problem domain due to the logical nature of the decision table semantics.

Finite sets C and A of conditions and actions and a number $n \in \mathbb{N}_0$ of rules constitute the learning assignment specification (cf. Figure 3). In Step (1), we define the signature of the SMT-problem as a set of constants, one constant per decision table cell. Over this signature, we generate formulae that, e.g., specify the allowed content of decision table cells and also the determinism property. In Step (2), an SMT-solver generates a model, i.e., an interpretation of the signature such that the considered formulae hold under this interpretation (if and only if a deterministic decision table of the desired size exists). In Step (3), we interpret the model as a decision table. To obtain a second, different deterministic decision table, the model from the previous run can be encoded as an additional formula. Then, taking Step (1) again, we obtain a model that encodes a different deterministic decision table (if and only if a second decision table of the desired form exists).

The idea of the encoding is as follows: For each premise cell of the desired decision table, there is one constant in the signature that can take values '×', '−', or '∗', and similarly for each effect cell, then with possible values '×' or '−'. So to obtain a table over m conditions and k actions with n rules (cf. Figure 1), we would have constants $v_{1,1}, \ldots, v_{m,n}$ and $w_{1,1}, \ldots, w_{k,n}$. Now any model of the trivial formula '$true$' as returned by an SMT-solver is already an encoding of a well-formed decision table: The entry of cell, e.g., $v_{i,j}$ takes the interpretation of the constant $v_{i,j}$, which is '×', '−', or '∗'.

Providing a constraint that corresponds to determinism is a bit more involved. Firstly, the definition of determinism (cf. (2)) refers to a fixed decision table with fixed premise formulae. For decision table generation, we need a formula that refers to the decision table *as encoded* by the constants $v_{i,j}$ and $w_{i,j}$. Secondly, the definition of determinism refers to validity of propositional formulae by using the validity operator '$\models$' (read: for each valuation of the conditions in C, the negation of two different rules' premise formula evaluates to $true$). Such a validity operator is usually not directly available in SMT-solvers.

We observe that the premise formula of a rule r in a decision table is obtained using the helper function F. Function F basically says, that a given valuation $\sigma : C \to \{0, 1\}$ of the conditions in C is enabling the i-th cell (in the row of condition c_i) of the j-th rule if and only if this cell holds symbol '×' and



(a) Generation of one decision table of size $(n+n) \times n$.

(b) Generation of a new bunch of 100 decision tables of size $(3+3) \times 3$.

Fig. 4: Runtimes for two kinds of decision table generation tasks (■: user time, ■: elapsed time, ■: kernel time).

$\sigma(c_i) = 1$, or this cell holds symbol '$-$' and $\sigma(c_i) = 0$, or this cell holds symbol '$*$' ("don't care"). In the SMT-encoding, the symbol in "this cell" is given by the value of constant $v_{i,j}$, hence the equivalence stated above corresponds to the formula

$$G_{i,j} := (v_{i,j} = \times \wedge c_i) \vee (v_{i,j} = - \wedge \neg c_i) \vee (v_{i,j} = *) \quad (3)$$

over a logical variable c_i. Formula (3) holds if and only if the current value of c_i enables the currently considered entry of the decision table cell as given by the interpretation of constant $v_{i,j}$. A rule r is enabled by a valuation σ if and only if σ satisfies $\mathcal{F}_{pre}(r)$, which is the case if and only if σ enables all cells that constitute the premise of r. Hence the formula $G_j := G_{1,j} \wedge \cdots \wedge G_{m,j}$ corresponds to the enabledness of rule r_j (where the rule is given by the current interpretation of the $v_{i,j}$). So to obtain a deterministic decision table, we conjoin the overall formula

$$G := \forall\, 1 \leq j_1 \neq j_2 \leq n \;\forall\, c_1, \ldots, c_m \bullet \neg(G_{j_1} \wedge G_{j_2}) \quad (4)$$

to the SMT-problem. The particular SMT-problem is now to provide an interpretation of the signature (i.e., the constants $v_{i,j}$ and $w_{i,j}$) such that G is satisfied. For details of the encoding, we refer the reader to [9]. Note that all quantifiers in G (cf. (4)) range over finite domains and can hence be expanded into a finite conjunction. The expanded formula is then treatable by efficient SMT-solvers for quantifier-free theories. The unfolding grows in the number of conditions and desired rules. Yet, since we aim to create decision tables that are supposed to be solved by humans, we expect that the unfolding of G into a quantifier-free formula is usually still well-treatable by today's SMT-solvers. Prelimary evaluation results as reported below confirm our expectation.

V. Evaluation

We have implemented our approach using the API of the SMT-solver SMTInterpol [10]. We imagine to offer to teachers a choice of decision tables for a given learning assignment specification, and the possibility to refine the specification, e.g., to limit the number of '$*$' symbols (the fewer '$*$' symbols a decision table has, the more obvious the determinism).

Figure 4 shows the runtime of decision table generation using our implementation. The practically most relevant graph is *elapsed time* (blue); the total *user time* (yellow) grows faster because the underlying solver effectively uses multi-core systems. The top graph shows the time needed to generate one decision table of size $(n+n)\times n$ (n conditions and actions, and n rules). For values of n up to 10, the runtime is in the order of seconds. Hence generating decision tables for our exercises and exams is well feasible, because our tasks usually have 3 to 5 conditions, about 3 actions, and not more than 7 rules. Working on larger decision tables in 'paper & pencil' style is in our opinion not supporting the learning but just tedious.

One use case that we envision for our procedure is to generate bunches of decision tables as task candidates for the teacher to choose from. The bottom graph in Figure 4 shows that the time needed for generating bunches of 100 decision tables for $n = 3$ is in average about half a second. An inspection of the bunches shows that using SMTInterpol has the effect that the tables in the bunches do not feel too regular (i.e., not like a mere enumeration of decision tables) and that we do not get too many symmetric decision tables. Still, we plan to give teachers the possibility to refine the specification, e.g., to limit the number of '$*$' symbols (the fewer '$*$' symbols a decision table has, the more obvious the determinism).

VI. Discussion

The previous sections show that it is possible to synthesise learning assignments for software description formalisms (viewed as mathematical objects) that satisfy certain precisely stated constraints. These constraints can be very generic, in our case the full power of logic over some theory is available. Here we see a strong potential of our approach: We *formalise* conditions that make learning assignments good learning assignments, relative to the teacher's intention and the students' situation in the same way as sketched for addition tasks in the introduction.

In the case of decision tables, we have for example observed that the tasks that we create anew each season for the exercises and the exam follow certain principles. One is of course the size of the tables (as already considered in our generation procedure). With non-deterministic decision tables, we do consider more aspects. For example, a table where two rules are just copies of each other could be too easy to analyse. We can immediately formalise that there are no two identical rules and add that constraint to the learning assignment specification. Then, our definition of determinism has the particularity that actions are not considered. So if two rules can be enabled by one valuation of the conditions and have the same effect, they are still (by definition) non-deterministic. A table with all different effects could be a good choice when starting with non-deterministic tables, yet the corner-case of overlapping premise and equivalent effect should be presented to the advanced learner. Again, we can formalise both cases as constraints. Furthermore, an analysis for non-determinism could be too easy if there are too many '$*$' symbols in the table, so we may want to add a constraint that limits the number of '$*$' symbols overall, or just per rule.

The same applies to other properties of decision tables, such as completeness, consistency, presence of useless rules, etc..

Once we have established a set of constraints that can be added to learning assignment generation runs, there are manifold applications. The most obvious application is to generate exercises or exam tasks tailored to a particular teaching or testing goal (cf. Figure 2). It is easy to embed our decision table synthesis procedure into a tool where a teacher only selects the kind of task to obtain a task statement and a matching decision table. We envision a tool that presents a selection of tasks to teachers. We feel that the teacher needs to decide and should not blindly rely on a tool. The experience from our experiments is that a set of 10 (or even 100) decision tables usually includes many examples of good exercises.

Having decision tables available in machine readable form allows us to generate much of the necessary material automatically, including correction schemes and slides for tutorial sessions. In tutorials that discuss non-determinism, there could be overlays for the case with disjoint effects and the corner-case with equivalent effects. Further applications could be to offer an e-learning tool, e.g., a website or a mobile application, that generates new learning assignments for a student-controlled or heuristically adapted level of difficulty. Note that we want to address learning activities that focus on the formalisms and analysis procedures as such. To this end, it is perfectly adequate that the generated decision tables do not model an existing system. In our opinion, students should not firstly assess the pragmatics of a decision table model, or try to look out for non-determinism in their imagination of the modelled system, but they should learn to rely on the purely abstract application of analysis procedures.

From our experience with exercises for our software engineering course, we anticipate that our approach can be extended to class diagrams, object diagrams, OCL constraints together with class diagrams, and even Statechart-like formalisms. It is an open research question which synthesis procedure is adequate for which formalism.

VII. Related Work

The idea to generate learning assignments is not new. worksheets for elementary school arithmetic are generated randomly (e.g., [11]) and, e.g., [12] use mutations of templates for state-machine problems. Singh et al. [13] infer a set of similar problems on algebraic equivalence proofs from an abstraction of a given problem into a so-called query, where queries can also serve to give learning assignment specifications. Similarly, Ahmed et al. [14] propose to abstract natural deduction proofs and to then generate comparable problems. Alvin et al. [15] targets the synthesis of geometry problems from an example with a set of properties to be preserved. They suggest that proof width and length, and the number of deductive steps are useful metrics to control the synthesis. Andersen et al. [16] propose to characterise the difficulty of problems by the execution trace of a procedure that solves the problem, and to synthesise problems along such a procedure to obtain a controlled learning progression. Sadigh et al. [12] provide a procedure to generate problems that ask to construct a DFA that accepts a certain language from a set of so-called seed problems. They propose a metric of difficulty based on number of states, depth, and number of counter-examples used by a learning algorithm.

Our approach can be seen to take one step back since we feel that teaching of software description languages is not yet as well-researched as, e.g., geometry or algebra. We put the teachers' understanding of learning assignments first and want to provide problem encodings such that teachers can formalise their understanding of problem classes and have complete control over generated problems.

VIII. Conclusion

We have presented a logic-based approach to characterise and synthesise learning assignments for a subset of the formal software description language of decision tables. A first evaluation shows that our procedure quickly generates problems of the size needed for 'paper & pencil' tasks. We argue to put computer science teachers into a prominent position when generating learning assignments: A formalisation of software engineering problems allows us teachers to formalise and investigate our understanding of good learning assignments.

Future research includes to extend the set of supported learning assignments (also for different formalisms), and further research into the nature of instructionally good learning assignments and their formal characterisation.

References

[1] N. M. Seel, *Lernaufgaben und Lernprozesse*, ser. Studienbuch Pädagogik. Stuttgart: W. Kohlhammer, 1981.

[2] A. Renkl, "Lernaufgaben zum Erwerb prinzipienbasierter Fertigkeiten," in *Lernaufgaben entwickeln, bearbeiten und überprüfen*, ser. Fachdidaktische Forschungen, vol. 6, 2014, pp. 12–22.

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

[4] B. Westphal, "An undergraduate requirements engineering curriculum with formal methods," in *REET@RE*, M. Moshirpour, M. Moussavi, A. M. Grubb, S. Gregory, and B. Far, Eds. IEEE, 2018, pp. 1–10.

[5] ——, "Formale methoden in der Softwaretechnik-Vorlesung," in *SEUH*, V. Thurner *et al.*, Eds., vol. 2358. CEUR-WS.org, 2019, pp. 21–33.

[6] ——, "Teaching software modelling in an undergraduate introduction to software engineering," in *EduSymp*. IEEE, 2019, pp. 0–0.

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

[8] ——, *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation, Betrieb*, 3rd ed. Spektrum, 2011.

[9] M. Steinle, "Automatische Generierung von Lernaufgaben für Entscheidungstabellen," 2019, B. Sc. thesis, Universität Freiburg.

[10] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An Interpolating SMT Solver," in *SPIN*, ser. LNCS, A. F. Donaldson and D. Parker, Eds., vol. 7385. Springer, 2012, pp. 248–254.

[11] mathworksheetwizard.com. (2019)

[12] D. Sadigh, S. A. Seshia, and M. Gupta, "Automating exercise generation: a step towards meeting the MOOC challenge for embedded systems," in *WESE*, P. Marwedel, Ed. ACM, 2012, p. 2.

[13] R. Singh, S. Gulwani, and S. K. Rajamani, "Automatically generating algebra problems," in *AAAI*, 2012.

[14] U. Z. Ahmed, S. Gulwani *et al.*, "Automatically generating problems and solutions for natural deduction," in *IJCAI*, 2013, pp. 1968–1975.

[15] C. Alvin, S. Gulwani, R. Majumdar, and S. Mukhopadhyay, "Automatic synthesis of geometry problems for an intelligent tutoring system," *CoRR*, vol. abs/1510.08525, 2015.

[16] E. Andersen *et al.*, "A trace-based framework for analyzing and synthesizing educational progressions," in *SIGCHI*. ACM, 2013, pp. 773–782.