The Education Column

by

Juraj Hromkovič and Dennis Komm

ETH Zürich, Switzerland

juraj.hromkovic@inf.ethz.ch and dennis.komm@inf.ethz.ch

Teaching Formal Foundations of Computer Science with Iltis

Marko Schmellenkamp
Ruhr University Bochum
marko.schmellenkamp@rub.de

Fabian Vehlken
Ruhr University Bochum
fabian.vehlken@rub.de

Thomas Zeume
Ruhr University Bochum
thomas.zeume@rub.de

Abstract

Introductory courses on formal foundations of computer science are often attended by large numbers of students with diverse backgrounds. In this paper we outline how we address this challenge in our courses by supplementing traditional teaching with web-based, interactive exercises. The web-based exercises are provided by Iltis, a modern teaching support system covering the foundations of computer science logic, formal languages, and (parts of) complexity theory. We give a gentle introduction to Iltis, describe its technical integration into our courses, and outline research challenges and opportunities coming up when developing such a system.

1 Introduction and Motivation

Formal foundations are at the core of many modern applications of computer science and are therefore an integral part in recommendations for Bachelor computer science curricula [8, 6]. Typical study programs implement these recommendations by offering, among others, courses on *Logics for Computer Scientists* — covering the reasoning pipeline for propositional logic and first-order logic — and on *Theoretical Foundations of Computer Science* — covering formal languages as well as basics of complexity and computability theory; see Figure 1(a) for an overview of standard topics.

Teaching these formal foundations is a challenge for most instructors as it is one of the harder topics for students. Also, increasing numbers of students enrolled in computer science courses with diverse backgrounds are difficult to

Foundations of logic

- Basics: Propositional, modal, and first-order logic
- Advanced: Logics for verification, description logics, etc.
- Methods: Modelling, reasoning pipeline (modelling, transformation, inference), algorithms for evaluation & satisfiability

Foundations of formal languages

- Basics: Regular and context-free languages
- Advanced: Regular tree languages, timed languages etc.
- Methods: Modelling, closure properties, constructions & algorithms, pumping lemmata

Basics of complexity theory

- Basics: P, NP, completeness
- Advanced: Space-based classes, fine-grained complexity, etc.
- Methods: Classifications, closure properties, reductions, . . .

Basics of computability theory

- Basics: (Semi-)decidability, undecidability, computability
- Methods: Classifications, closure properties, reductions, . . .

(a)

Exercise: From modelling to inference

Julia has identified the following dependencies between program libraries and system libraries:

- No software package is both a program library and a system library.
- System libraries depend only on system libraries.
- Every software package that must be explicitly installed by the user depends on at least one program library directly.

She concludes that there is no system library that must be explicitly installed by the user.

Can you confirm her conclusion using methods you learned for first-order logic?

(b)

Figure 1: (a) Topics typically covered by courses on formal foundations of computer science. (b) A typical exercise for a workflow covering modelling, transformation, and inference in first-order logic.

handle with traditional lecture- and tutorial-based courses. In particular, providing individual human tutoring for such a large number of students with diverse needs exceeds the resources of most CS departments. A general approach for tackling this challenge and increasing learning outcomes across STEM disciplines is provided by the National Research Council of the US which advocates, among others, to "Leverage technologies to make the most effective use of students' time, shifting from information delivery to sense-making and practice in class" [9, 2]. For formal foundations, technological teaching support in particular may help to make room for theory and in-depth problem solving in lectures and tutorials by outsourcing some basics.

From our perspective, to be useful in large, mandatory courses, teaching support technologies for formal foundations of computer science need to offer:

- *Coverage* of a wide range of topics in formal foundations of computer science;
- *Advanced feedback and support* provided immediately, extensively, and individually;
- *Flexibility* in how to use and combine educational tasks;
- *Easy integration* into courses; and
- *Extensibility* of topical range and feedback mechanisms.

Many teaching support systems for topics typically taught in introductory formal foundations courses have been developed over the years. Most of these systems were developed ad-hoc by instructors for helping their students. A common theme is that only a small set of topics (typically only one) is covered; systems are abandoned and/or become technologically outdated rather quickly; and in most of them only very basic feedback is provided.

In this article we report on our experiences and progress in building the teaching support system Iltis.[1] In short, Iltis offers a wide range of interactive, web-based exercises on formal foundations of computer science. It is designed for flexibility and extensibility, and offers easy integration into common learning management systems. Within this article we address

- the scope of Iltis – which topics are covered and how content can be composed for different needs (see Section 2);
- how we set up large introductory courses on *Logic for Computer Science* and on *Foundations of Theoretical Computer Science* with integrated web-based exercises in Iltis (see Section 3);
- what research challenges and opportunities arise – theoretical, practical, and didactical – when building teaching support systems for formal foundations of computer science (see Section 4).

This article updates, adapts, and condenses a report from 2021 by a superset of the current authors [5].

2 An Introduction to Iltis

In Iltis, instructors can design educational content flexibly by using a broad portfolio of educational tasks in foundations of logic, formal languages, and complexity theory (see Section 2.3). A compositional task model allows to combine tasks flexibly into multi-step exercises (see Section 2.1). A compositional feedback model allows for providing feedback according to the progress of students in curricula (see Section 2.2).

2.1 Compositional Task Model

Exercises in Iltis are built from small, easily composable, educational tasks. Each educational task is configurable by inputs — either given explicitly or as the output of prior tasks — and provides objects created by students within this task as outputs. The outputs can then be used by subsequent educational tasks. For instance, a

[1]Iltis [ˈɪltɪs] is the German word for polecat and the Swiss animal of the year 2024 [1]. We invite all readers to try out Iltis: `https://iltis.cs.tu-dortmund.de/`

Table 1: A summary of educational tasks in the logic domain that are supported by Iltis.

Task	Propositional logic	Modal logic	First-order logic
Evaluating formulas	✓	✓	–
Constructing models	✓	✓	✓
Creating signatures	✓	✓	(✓)
Constructing formulas	✓	✓	✓
Transforming	✓	✓	✓
Testing satisfiability	✓	✓	✓
Task variants & further tasks	Satisfiability tests with • truth tables • HornSat algorithm • tableau calculus • resolution	Satisfiability test with tableau calculus Calculating bisimulations Proving non-bisimilarity of worlds	Satisfiability test with resolution Proving non-equivalence of formulas

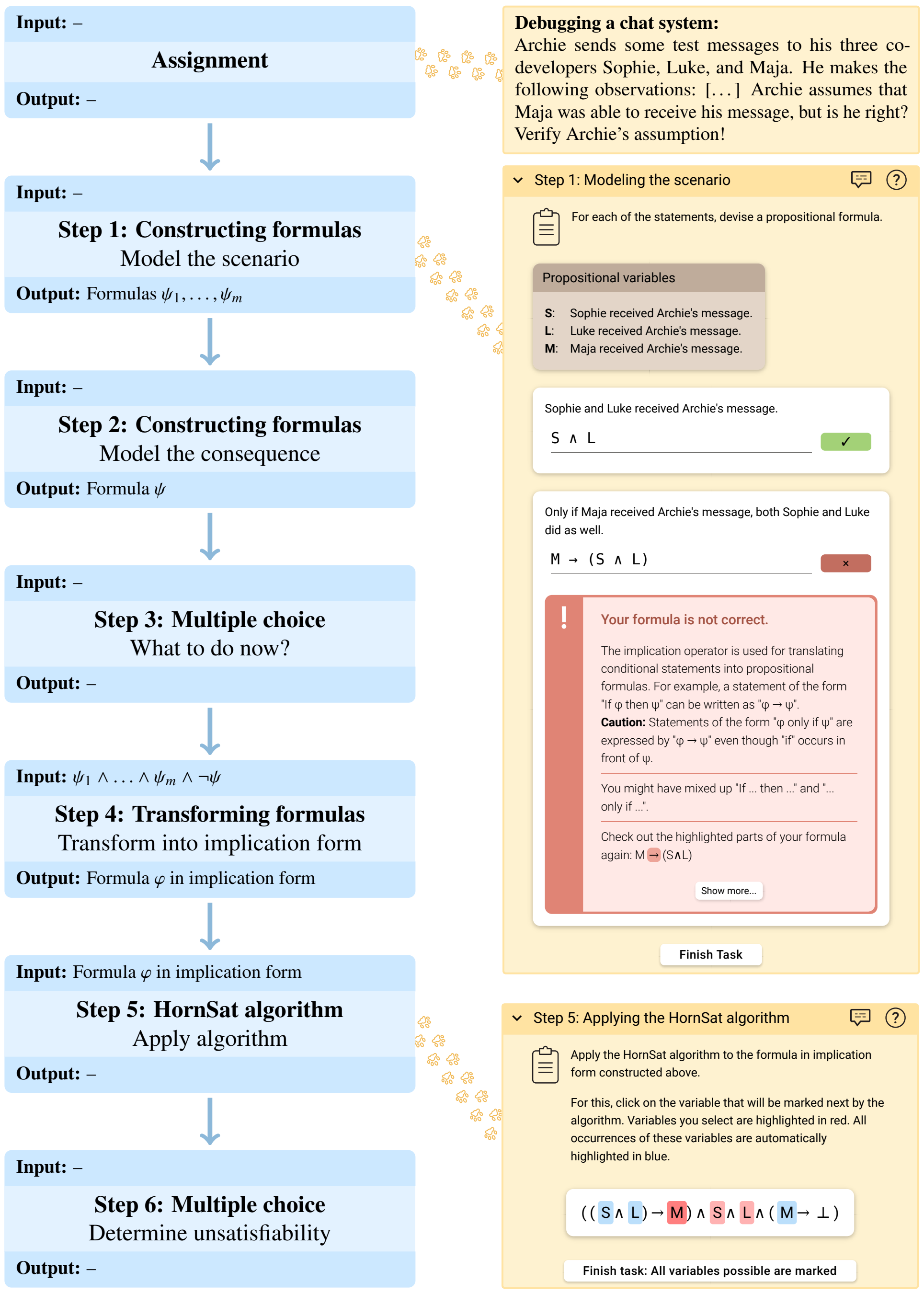
Table 2: A summary of educational tasks that are supported for formal languages, computability and complexity theory.

Regular languages	Context-free languages	Computability & complexity theory
Modeling with • deterministic automata • non-deterministic automata • regular expressions Specifying words Specifying Myhill-Nerode-classes Proving non-equivalence of languages	Modeling with • push-down automata • deterministic push-down automata • context-free grammars (CFGs) Specifying words Specifying derivations in CFGs Proving non-equivalence of languages	Interacting with graphs: • constructing graphs • colouring nodes and edges satisfying multiple conditions Specifying graph reductions

task for transforming a formula into conjunctive normal form (CNF) receives a formula as input and provides the student-constructed, equivalent formula in CNF as output.

Typical workflows used in formal foundations of computer science can be covered by multi-step exercises composed of different educational tasks. For instance, Figure 2 illustrates a workflow in which students first model a scenario with propositional formulas φ_1 and φ_2, and then infer another propositional formula ψ by first deciding what to do, then transforming $\varphi_1 \wedge \varphi_2 \wedge \neg\psi$ into CNF, and finally showing unsatisfiability of the set of clauses via the satisfiability algorithm for Horn formulas (instead of the latter, also propositional resolution or the propositional tableau calculus could be used). Throughout this workflow, the formulas entered by the students are used for subsequent tasks.

Two further multi-step exercises are illustrated in Figures 2, 3 and 5.



Input: –
Assignment
Output: –
Input: –
Step 1: Constructing formulas
Model the scenario
Output: Formulas $\psi_1, \ldots, \psi_m$
Input: –
Step 2: Constructing formulas
Model the consequence
Output: Formula ψ
Input: –
Step 3: Multiple choice
What to do now?
Output: –
Input: $\psi_1 \wedge \ldots \wedge \psi_m \wedge \neg\psi$
Step 4: Transforming formulas
Transform into implication form
Output: Formula φ in implication form
Input: Formula φ in implication form
Step 5: HornSat algorithm
Apply algorithm
Output: –
Input: –
Step 6: Multiple choice
Determine unsatisfiability
Output: –

Debugging a chat system:
Archie sends some test messages to his three co-developers Sophie, Luke, and Maja. He makes the following observations: [...] Archie assumes that Maja was able to receive his message, but is he right? Verify Archie's assumption!

Step 1: Modeling the scenario

For each of the statements, devise a propositional formula.

Propositional variables

S:	Sophie received Archie's message.
L:	Luke received Archie's message.
M:	Maja received Archie's message.

Sophie and Luke received Archie's message.

S ∧ L ✓

Only if Maja received Archie's message, both Sophie and Luke did as well.

M → (S ∧ L) ×

Your formula is not correct.

The implication operator is used for translating conditional statements into propositional formulas. For example, a statement of the form "If φ then ψ" can be written as "φ → ψ". **Caution:** Statements of the form "φ only if ψ" are expressed by "φ → ψ" even though "if" occurs in front of ψ.

You might have mixed up "If ... then ..." and "... only if ...".

Check out the highlighted parts of your formula again: M → (S∧L)

Show more...

Finish Task

Step 5: Applying the HornSat algorithm

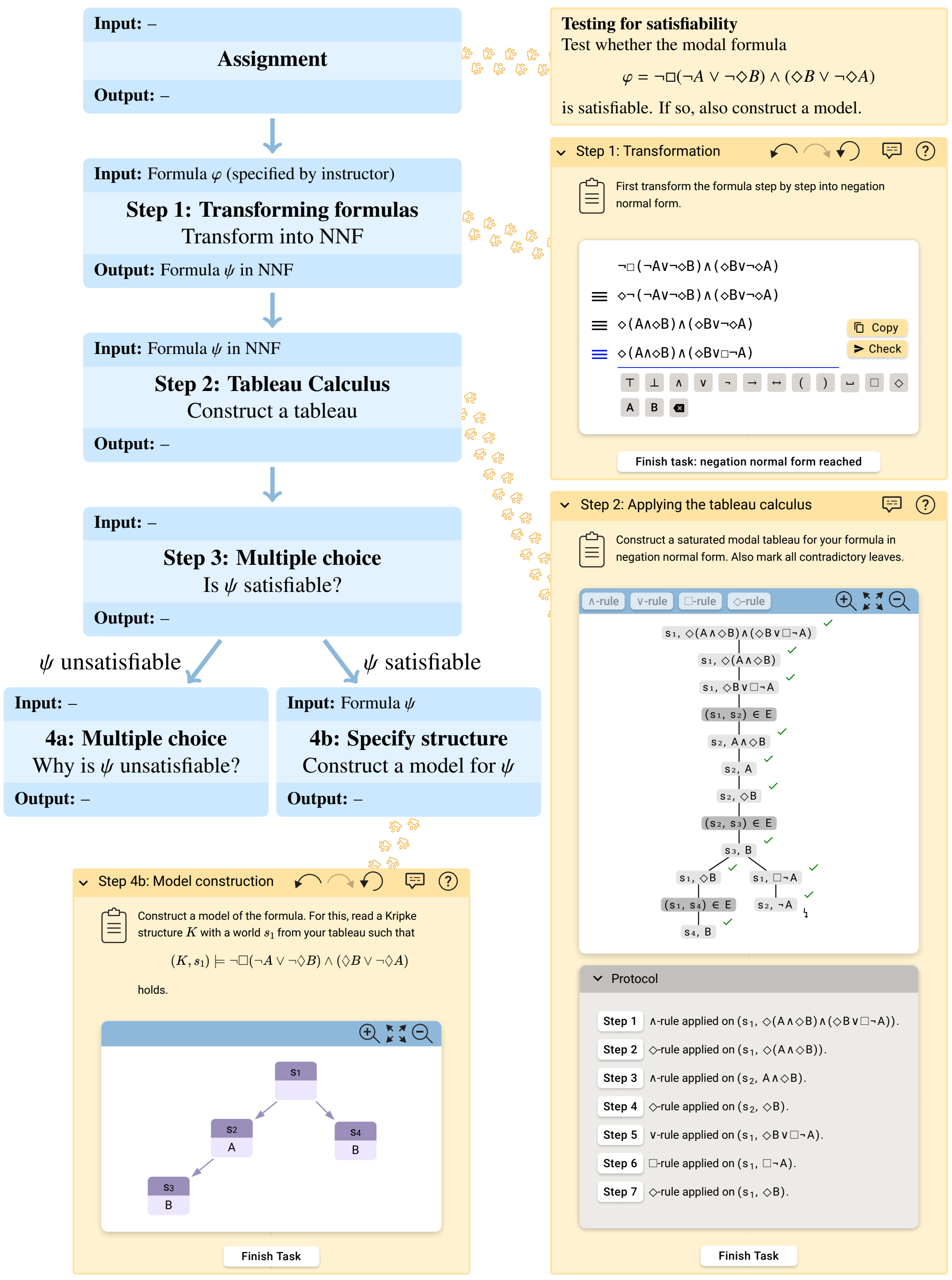
Apply the HornSat algorithm to the formula in implication form constructed above.

For this, click on the variable that will be marked next by the algorithm. Variables you select are highlighted in red. All occurrences of these variables are automatically highlighted in blue.

((S ∧ L) → M) ∧ S ∧ L ∧ (M → ⊥)

Finish task: All variables possible are marked

Figure 2: An exercise for the propositional reasoning workflow, composed of smaller educational tasks. For this sample scenario, the instructor chose the HornSat satisfiability test as Horn formulas are sufficiently expressive. For general propositional formulas, also truth tables, propositional resolution, and the propositional tableau calculus can be used. In Step 1, the student chose to reveal the first three feedbacks.



Testing for satisfiability
Test whether the modal formula

$$\varphi = \neg\Box(\neg A \lor \neg\Diamond B) \land (\Diamond B \lor \neg\Diamond A)$$

is satisfiable. If so, also construct a model.

First transform the formula step by step into negation normal form.

$\neg\Box(\neg A \lor \neg\Diamond B) \land (\Diamond B \lor \neg\Diamond A)$

$\equiv \Diamond\neg(\neg A \lor \neg\Diamond B) \land (\Diamond B \lor \neg\Diamond A)$

$\equiv \Diamond(A \land \Diamond B) \land (\Diamond B \lor \neg\Diamond A)$

$\equiv \Diamond(A \land \Diamond B) \land (\Diamond B \lor \Box\neg A)$

Construct a saturated modal tableau for your formula in negation normal form. Also mark all contradictory leaves.

$s_1, \Diamond(A \land \Diamond B) \land (\Diamond B \lor \Box\neg A)$ ✓
$s_1, \Diamond(A \land \Diamond B)$ ✓
$s_1, \Diamond B \lor \Box\neg A$ ✓
$(s_1, s_2) \in E$
$s_2, A \land \Diamond B$ ✓
s_2, A ✓
$s_2, \Diamond B$ ✓
$(s_2, s_3) \in E$
s_3, B ✓
$s_1, \Diamond B$ ✓ | $s_1, \Box\neg A$ ✓
$(s_1, s_4) \in E$ | $s_2, \neg A$ ✓ ↯
s_4, B ✓

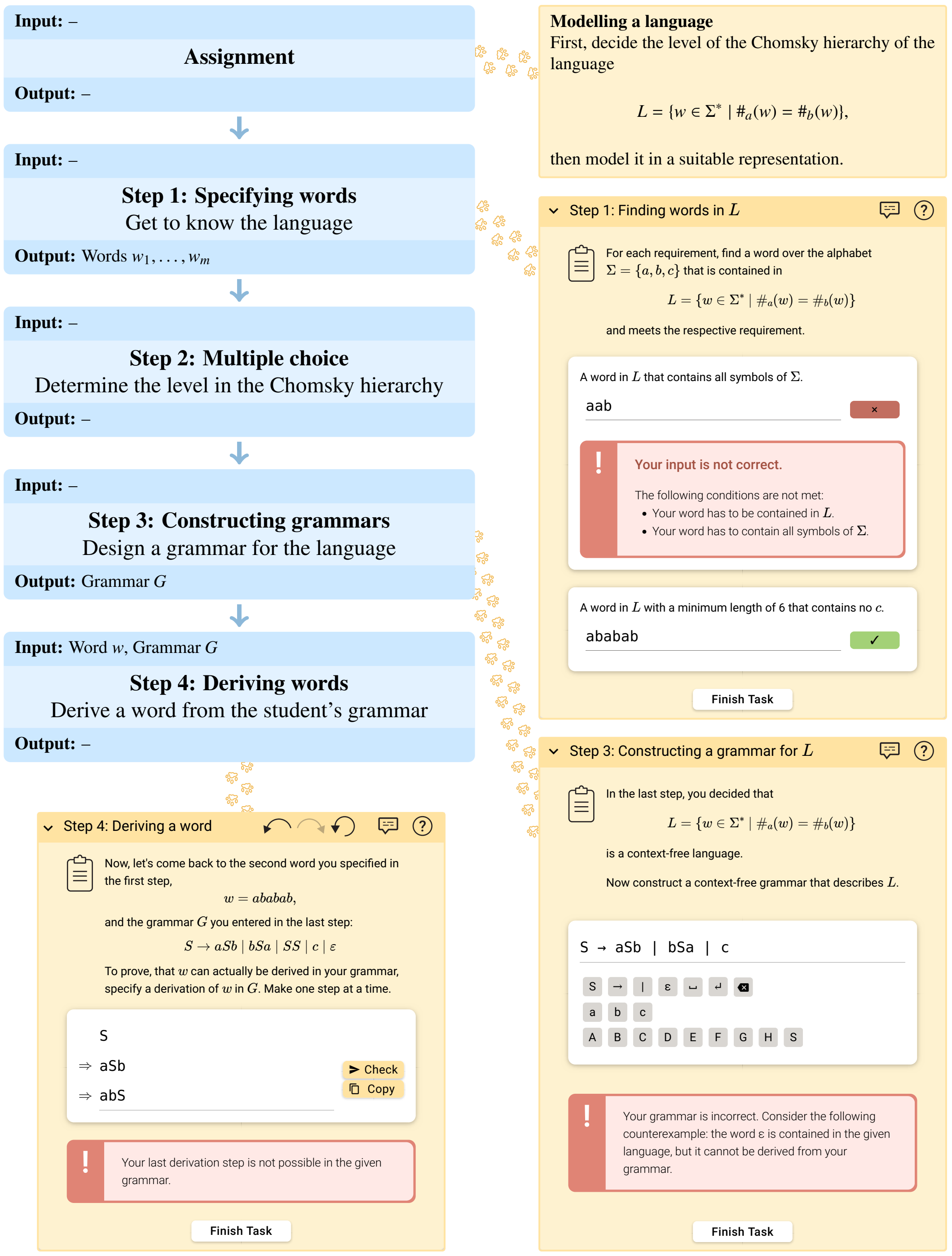
Step 1	∧-rule applied on $(s_1, \Diamond(A \land \Diamond B) \land (\Diamond B \lor \Box\neg A))$.
Step 2	◇-rule applied on $(s_1, \Diamond(A \land \Diamond B))$.
Step 3	∧-rule applied on $(s_2, A \land \Diamond B)$.
Step 4	◇-rule applied on $(s_2, \Diamond B)$.
Step 5	∨-rule applied on $(s_1, \Diamond B \lor \Box\neg A)$.
Step 6	□-rule applied on $(s_1, \Box\neg A)$.
Step 7	◇-rule applied on $(s_1, \Diamond B)$.

Construct a model of the formula. For this, read a Kripke structure K with a world s_1 from your tableau such that

$$(K, s_1) \models \neg\Box(\neg A \lor \neg\Diamond B) \land (\Diamond B \lor \neg\Diamond A)$$

holds.

Figure 3: An exercise for solving the satisfiability problem for modal formulas, composed of smaller educational tasks. For satisfiable and unsatisfiable formulas, different workflows can be used.



Input: –

Assignment

Output: –

Input: –

Step 1: Specifying words
Get to know the language

Output: Words $w_1, \dots, w_m$

Input: –

Step 2: Multiple choice
Determine the level in the Chomsky hierarchy

Output: –

Input: –

Step 3: Constructing grammars
Design a grammar for the language

Output: Grammar G

Input: Word w, Grammar G

Step 4: Deriving words
Derive a word from the student's grammar

Output: –

Modelling a language
First, decide the level of the Chomsky hierarchy of the language

$$L = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\},$$

then model it in a suitable representation.

Step 1: Finding words in L

For each requirement, find a word over the alphabet $\Sigma = \{a, b, c\}$ that is contained in

$$L = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$$

and meets the respective requirement.

A word in L that contains all symbols of Σ.

aab ×

Your input is not correct.

The following conditions are not met:
- Your word has to be contained in L.
- Your word has to contain all symbols of Σ.

A word in L with a minimum length of 6 that contains no c.

ababab ✓

Finish Task

Step 3: Constructing a grammar for L

In the last step, you decided that

$$L = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$$

is a context-free language.

Now construct a context-free grammar that describes L.

```
S → aSb | bSa | c
```

Your grammar is incorrect. Consider the following counterexample: the word ε is contained in the given language, but it cannot be derived from your grammar.

Finish Task

Step 4: Deriving a word

Now, let's come back to the second word you specified in the first step,

$$w = ababab,$$

and the grammar G you entered in the last step:

$$S \to aSb \mid bSa \mid SS \mid c \mid \varepsilon$$

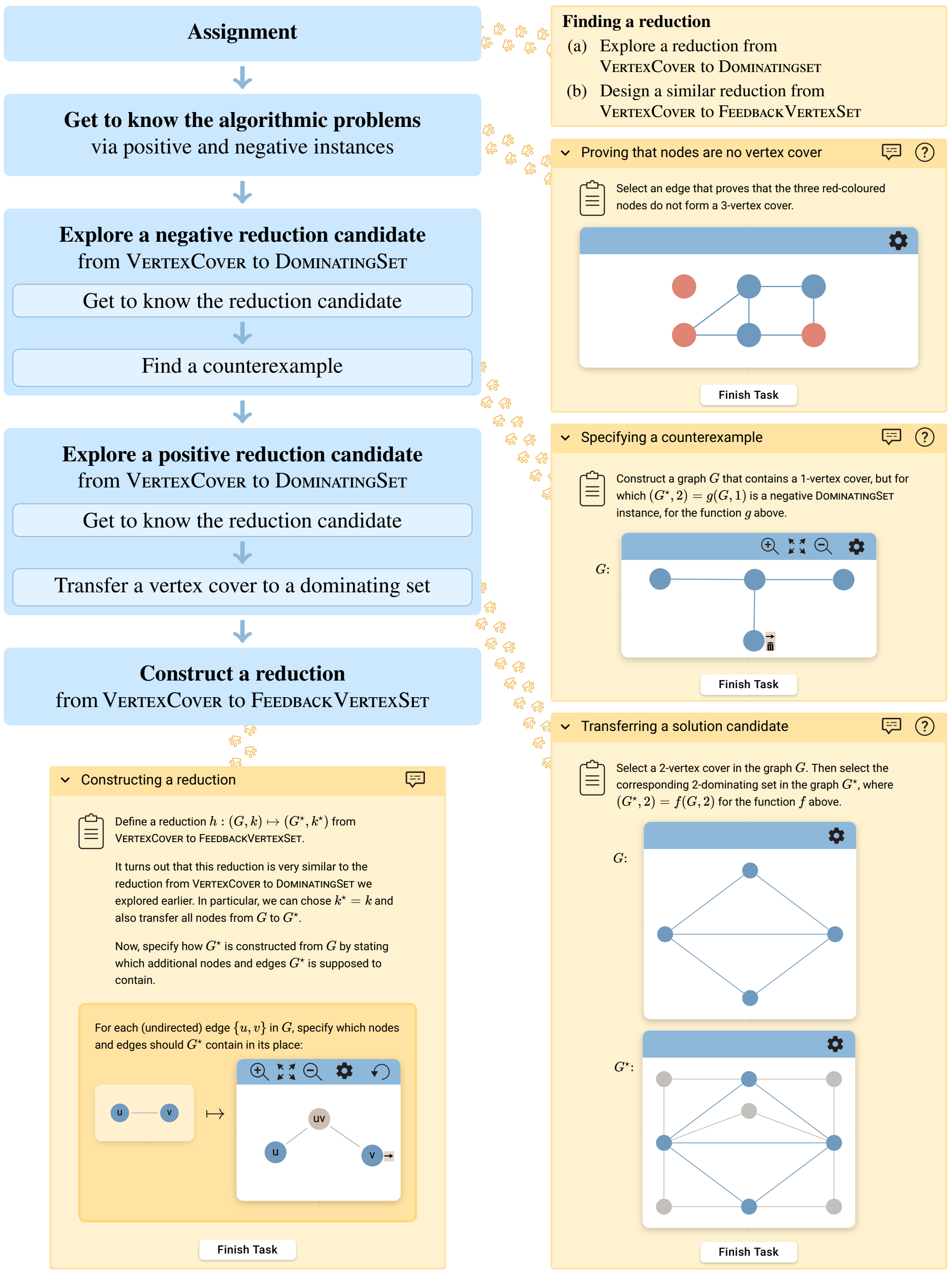
To prove, that w can actually be derived in your grammar, specify a derivation of w in G. Make one step at a time.

```
  S
⇒ aSb
⇒ abS
```

Your last derivation step is not possible in the given grammar.

Finish Task

Figure 4: An exercise for constructing a representation of a formal language. As preparatory step, students explore the formal language by identifying some elements and deciding its level in the Chomsky hierarchy. For constructing a representation of a context-free languages Iltis supports context-free grammars and push-down automata; for regular languages, it supports (non-)deterministic finite state automata and regular expressions.


Assignment
Get to know the algorithmic problems via positive and negative instances
Explore a negative reduction candidate from VertexCover to DominatingSet
Get to know the reduction candidate
Find a counterexample
Explore a positive reduction candidate from VertexCover to DominatingSet
Get to know the reduction candidate
Transfer a vertex cover to a dominating set
Construct a reduction from VertexCover to FeedbackVertexSet
Constructing a reduction
Finish Task
Proving that nodes are no vertex cover
Finish Task
Specifying a counterexample
G:
Finish Task
Transferring a solution candidate
G:
G^*:
Finish Task
u
v
uv

Finding a reduction

(a) Explore a reduction from VertexCover to DominatingSet

(b) Design a similar reduction from VertexCover to FeedbackVertexSet

Select an edge that proves that the three red-coloured nodes do not form a 3-vertex cover.

Construct a graph G that contains a 1-vertex cover, but for which $(G^*, 2) = g(G, 1)$ is a negative DominatingSet instance, for the function g above.

Select a 2-vertex cover in the graph G. Then select the corresponding 2-dominating set in the graph G^*, where $(G^*, 2) = f(G, 2)$ for the function f above.

Define a reduction $h : (G, k) \mapsto (G^*, k^*)$ from VertexCover to FeedbackVertexSet.

It turns out that this reduction is very similar to the reduction from VertexCover to DominatingSet we explored earlier. In particular, we can chose $k^* = k$ and also transfer all nodes from G to G^*.

Now, specify how G^* is constructed from G by stating which additional nodes and edges G^* is supposed to contain.

For each (undirected) edge $\{u, v\}$ in G, specify which nodes and edges should G^* contain in its place:

Figure 5: A series of tasks for helping students to find a computational reduction between two problems. As a preparatory step, students can explore (a) the involved algorithmic problems and (b) reduction candidates similar to the reduction to be found (one incorrect and one correct candidate). The actual workflow is only depicted partially in this illustration.

2.2 Compositional Feedback Model

One of the core objectives of Iltis is to provide immediate and comprehensive feedback, as this is one of the most important factors for learning success. Educational task types in Iltis come with multiple *feedback generators*, each one responsible for one kind of feedback. Individual feedback generators can be composed to *feedback strategies* by simple rule-based programs. Such programs are executed upon student input and determine the order of feedback, with the execution of rules possibly depending on the result of previous rules.

When specifying interactive exercises, instructors can state which feedback strategy to use (or they can define a custom one). In this way, the progress of students can be taken into account, e.g., a strategy that provides a lot of feedback can be used for beginners, while a strategy that provides almost no feedback can be used for exam preparation. By gradually uncovering the feedback from the different generators, students can choose how much of the feedback provided they want to use.

Designing feedback strategies and generators is a subtle and challenging task as algorithmic feasibility as well as didactical aspects have to be taken into account. We sketch a sample strategy and its feedback generators for providing feedback for the construction of propositional formulas (see Figure 2, Step 1, for a partial illustration):

(1) *Correctness:* Is the constructed formula *correct* or *not correct*?

(2) *Misconceptions:* Typical misconceptions (e.g., mixing up premise and conclusion of an implication, especially when modelling "only if"-statements) are identified using an abstract rule framework [4]. They are the basis for several feedback generators:

   (a) *Hint at the misconception* (e.g., "Do you remember how 'if'- and 'only if'-statements can be expressed in propositional logic?")
   (b) *State the misconception explicitly* (e.g., "You might have mixed up 'if' and 'only if'.")
   (c) *Point out the precise position of the mistake*

(3) *Distinguishing model:* A valuation that distinguishes the constructed formula from a correct formula.

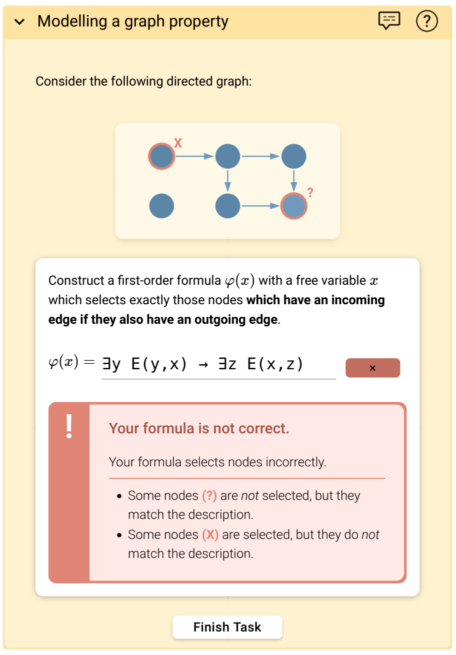
2.3 Educational Tasks

Iltis supports a variety of educational tasks for foundations of computer science logic, formal languages, and (parts of) complexity theory. In addition to content-specific tasks, there are also tasks to smooth out multi-step exercises, such as multiple-choice tasks.

Foundations of logic. Educational tasks for foundations of logic cover propositional logic, modal logic, and first-order logic content (see Table 1 for an overview). A broad spectrum of typical tasks is covered, including:

- *Evaluating formulas:* Students can evaluate formulas for a given interpretation by constructing truth tables for propositional formulas or evaluation tables for a given modal formula and Kripke structure, respectively.
- *Constructing models:* Students can construct models (i.e., satisfying interpretations) for formulas. For propositional logic, models are specified by valuations for all variables, for modal logic by Kripke structures; and for first-order logic, students can construct structures.
- *Creating signatures:* Students can specify a suitable logical language for describing a scenario. For propositional and modal logic, students can specify which propositional variables they want to use and describe their meaning in natural language (see Figure 7). For first-order logic, this is currently being implemented. A prototype for describing first-order signatures in natural language is available.
- *Constructing formulas:* Students can construct formulas for natural-language descriptions of scenarios provided by the instructor. For first-order logic, there is a second variant where teachers can provide a textual description of a unary graph query as well as a sample graph, and students are asked to provide a first-order formula that selects the same nodes as the graph query on this sample graph (see Figure 6 for both variants).
- *Applying equivalence transformations:* Students can transform propositional, modal, and first-order formulas step-by-step either into an equivalent target formula or into an equivalent formula in a given normal form.
- *Testing satisfiability:* Students can test formulas for satisfiability using several methods such as resolution or the tableau calculus.

Foundations of formal languages. Educational tasks for foundations of formal languages cover regular languages and context-free languages (see Table 2 for an overview):



Modelling a graph property

Consider the following directed graph:

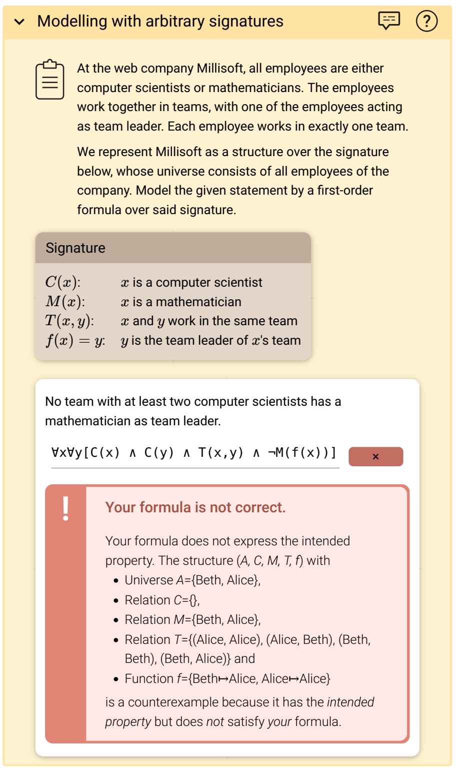
Construct a first-order formula $\varphi(x)$ with a free variable x which selects exactly those nodes **which have an incoming edge if they also have an outgoing edge**.

$\varphi(x) =$ `∃y E(y,x) → ∃z E(x,z)`

! Your formula is not correct.

Your formula selects nodes incorrectly.

- Some nodes (?) are *not* selected, but they match the description.
- Some nodes (X) are selected, but they do *not* match the description.



Modelling with arbitrary signatures

At the web company Millisoft, all employees are either computer scientists or mathematicians. The employees work together in teams, with one of the employees acting as team leader. Each employee works in exactly one team.

We represent Millisoft as a structure over the signature below, whose universe consists of all employees of the company. Model the given statement by a first-order formula over said signature.

Signature

$C(x)$:	x is a computer scientist
$M(x)$:	x is a mathematician
$T(x, y)$:	x and y work in the same team
$f(x) = y$:	y is the team leader of x's team

No team with at least two computer scientists has a mathematician as team leader.

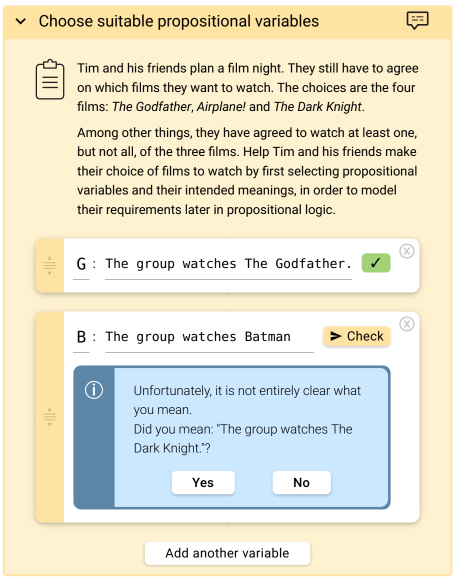
`∀x∀y[C(x) ∧ C(y) ∧ T(x,y) ∧ ¬M(f(x))]`

! Your formula is not correct.

Your formula does not express the intended property. The structure (*A*, *C*, *M*, *T*, *f*) with
- Universe *A*={Beth, Alice},
- Relation *C*={},
- Relation *M*={Beth, Alice},
- Relation *T*={(Alice, Alice), (Alice, Beth), (Beth, Beth), (Beth, Alice)} and
- Function *f*={Beth↦Alice, Alice↦Alice}

is a counterexample because it has the *intended property* but does *not* satisfy *your* formula.

Figure 6: Educational tasks for constructing first-order formulas over graph signatures (left) and general signatures (right).



Choose suitable propositional variables

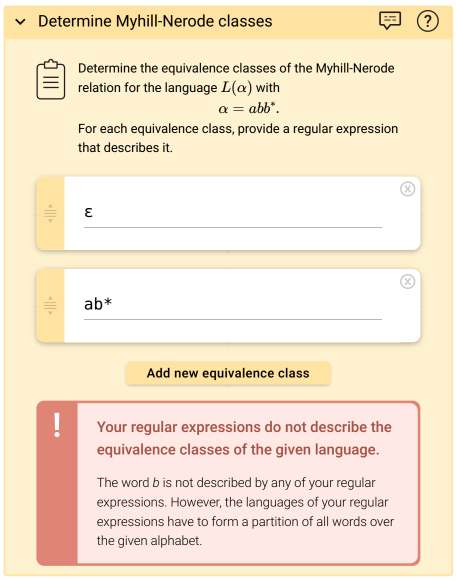
Tim and his friends plan a film night. They still have to agree on which films they want to watch. The choices are the four films: *The Godfather*, *Airplane!* and *The Dark Knight*.

Among other things, they have agreed to watch at least one, but not all, of the three films. Help Tim and his friends make their choice of films to watch by first selecting propositional variables and their intended meanings, in order to model their requirements later in propositional logic.

`G : The group watches The Godfather.` ✓

`B : The group watches Batman` ➤ Check

Unfortunately, it is not entirely clear what you mean.
Did you mean: "The group watches The Dark Knight."?



Determine Myhill-Nerode classes

Determine the equivalence classes of the Myhill-Nerode relation for the language $L(\alpha)$ with
$\alpha = abb^*$.
For each equivalence class, provide a regular expression that describes it.

`ε`

`ab*`

! Your regular expressions do not describe the equivalence classes of the given language.

The word *b* is not described by any of your regular expressions. However, the languages of your regular expressions have to form a partition of all words over the given alphabet.

Figure 7: Educational task for specifying propositional variables and their intended meaning. The intended meaning is verified via natural language processing models (currently fine-tuned for German exercises).

Figure 8: Educational task for identifying the Myhill-Nerode classes of a language.

- *Modelling:* Students can model languages with a variety of representations. For regular languages, (non-)deterministic automata and regular expressions can be used. For context-free languages, deterministic and general push-down automata and context-free grammars can be used.
- *Specifying Myhill-Nerode classes:* Students can specify regular expressions for the equivalence classes of the Myhill-Nerode relation.
- *Specifying derivations:* Students can derive a given word from a given context-free grammar step-by-step. For each step, Iltis checks whether it is valid.
- *Specifying words:* Students can specify words over a given alphabet. Then, Iltis checks whether they are contained in given (combinations of) regular or context-free languages. In this way, students can also prove the non-equivalence of languages.

Foundations of complexity theory. Educational tasks for foundations of complexity theory and computability theory are currently under development. In complexity theory, educational tasks for understanding algorithmic problems and computational reductions are already covered:

- *Interacting with graphs:* For understanding graph problems, students can select and colour nodes and edges in given graphs and build new graphs from scratch. The user input can be tested for a variety of conditions. This educational task can be used very flexibly.
- *Specifying graph reductions:* Students can specify certain graph reductions by specifying in a modular way how to map nodes and edges from an instance of the source problem to an instance of the target problem.

3 Instructor's Perspective: A Course Set-Up

We integrated web-based exercises provided by Iltis into our courses *Logic in Computer Science* (Bachelor, mandatory, 2 + 1 contact hours)[2] and *Foundations of Theoretical Computer Science* (Bachelor, mandatory, 4 + 2 contact hours)[3] at Ruhr University Bochum, each with > 200 students. The exercises are also used in similar courses at TU Dortmund University with > 400 students.

[2]The web-based exercises for *Logic in Computer Science* can be found at `https://iltis.cs.tu-dortmund.de/Logic-external/de/`

[3]The web-based exercises for *Foundations of Theoretical Computer Science* can be found at `https://iltis.cs.tu-dortmund.de/TCS-external/de/`

Course organisation. The set-up of the courses differs slightly, we focus on the *Foundations of Theoretical Computer Science* course covering regular languages, context-free languages, an introduction to computability theory, and an introduction to complexity theory. Organisation-wise, the course consists of:

- *Lectures:* Two traditional 90-minute lectures per week with all students, interrupted by few questions testing understanding via a student classroom response system.

- *Tutorials:* One 90-minute tutorial per week in groups of 20–30 students. First half spent on active problem solving in small groups and discussion. Second half spent on discussing solutions to assignments.

- *Assignments:* Consisting of (a) interactive web-based exercises provided by Iltis, some graded and some ungraded; (b) traditional assignments graded by tutors. The interactive web-based exercises are typically easier and students can try as often as they want, receiving feedback from Iltis for each attempt.

Our objective for the integration of interactive, web-based exercises was two-fold. First, to offer students the opportunity to train basics and receive feedback very early on. Having understood the basics, they then go on to tutorials and to the more complex traditional assignments. Second, outsourcing the basics to Iltis helps to save valuable time of teaching assistants, which then can be used to discuss more difficult topics and for in-depth problem solving in tutorials. The web-based exercises are provided at the time of the lecture and our recommendation for students is to do the web-based exercises before going to tutorials and starting with the analog exercises.

Technical organisation. The assignments – web-based and analog – are managed through our universities learning management platform Moodle. Both web-based and analog exercises have a digital twin in Moodle. The grading of the web-based exercises is handled by Iltis; the grading of the analog exercises by teaching assistants. The accounting of points is handled by Moodle.

Iltis exercises are integrated into the learning management platform via the LTI standard [10], which is supported by many modern teaching management platforms. They are stored in easily configurable XML files which are managed via Git repositories. Content for new courses can be created by starting from a clone of an existing course, selecting from a portfolio of existing exercises and adapting them according to the requirements.

4 Research Challenges & Opportunities

Building teaching support systems for formal foundations of computer science comes with a multitude of challenges and research opportunities. We sketch some of them.

A theory challenge

Providing teaching support for formal foundations of computer science – including feedback and advice for students and learning analytics for instructors – requires to solve algorithmic problems that are in most cases provably algorithmically hard or even unsolvable.

The main road block for teaching support systems for formal foundations is that many of the algorithmic tasks that need to be solved are inherently hard or even algorithmically unsolvable in general. For instance, deciding whether a scenario has been correctly modelled by a first-order formula is algorithmically impossible in general, due to the undecidability of testing whether two first-order formulas have the same meaning. While this is possible for propositional logic, no efficient algorithm is known so far. The same algorithmic hardness holds for many educational tasks for the foundations of logics, formal languages, complexity theory, and computability theory. Providing feedback beyond the mere fact whether a solution is correct – i.e., feedback that hints at the students' misconceptions or gently guides students to correct solutions – is potentially even harder.[4]

Attacking this challenge requires to find creative approaches for circumventing algorithmic hardnesses and often leads to interesting theoretical research questions. Hope for successfully tackling this challenge is provided by the fact that human tutors manage to provide feedback and advice.

We sketch two concrete examples, where methods from theoretical computer science helped us to come up with approaches for providing meaningful feedback and advice for students:

- **Explaining mistakes in context-free grammars.** In ongoing work on providing explanations for mistakes in modelling with context-free formalisms, we aim for feedback along the lines of the following interaction.

  Assignment: *Design a context-free grammar for* $L = \{a^n b^{n+2} \mid n \in \mathbb{N}\}$.

  Student: $S \to aSb \mid abb$

  Feedback (provided in customizable stages):

  (1) Correctness: *Your grammar is not correct.*

[4]The web-based teaching support system *AutomataTutor* provides advanced, high-level feedback for finite automata constructed by students [3].

(2) Pinpointing mistakes:

(a) Hinting at a wrong language: *Your grammar describes the language*

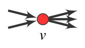
$$L = \{a^n b^{n+1} \mid n \in \mathbb{N}\}.$$

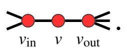
(b) Hinting at a wrong rule: *It might help to have a look at the rule*

$$S \to abb.$$

(c) Provide a counterexample: *The language described by your grammar contains the word abb, which is not in L.*

Even though testing correctness of student-provided solutions is undecidable in general, the above explanations can be provided efficiently by combining theory for bounded context-free languages [7], canonization of grammars, and grammar transformations.

- **Designing computational reductions.** For teaching reductions, instructors often design tasks for (i) understanding the computational problems involved, (ii) exploring existing reductions via examples, and (iii) designing reductions between computational problems. Task (iii) is a challenge for most students as the design of reductions usually does not follow a straightforward path but requires some creativity by students. When looking for a reduction, one approach by a typical expert is to sequentially try a number of building blocks that they have encountered in the context of other reductions before. An example is provided by the standard reduction from the problem of finding a directed Hamilton path to finding an undirected Hamilton path that transforms a directed graph to an undirected graph by mapping each node 

to a small gadget 

. Constructing such gadgets is one of the typical building blocks when designing reductions.

  One possible approach for supporting instructors in teaching how to design reductions, is to (a) identify and formalize typical building blocks of reductions, (b) develop a simple descriptional language for reductions that allows for combining the building blocks in a simple, modular manner, to (c) study the expressive power of such a language as well as the computational problems arising in finding reductions in such languages, and finally (d) for designing suitable feedback mechanisms for student attempts on constructing reductions. The exploration of (a)–(d) lead to interesting theoretical problems and has already been prototypically implemented (see Figure 5).

An engineering challenge

Building a flexible, extensible, usable, and maintainable teaching support system for formal foundations of computer science is a complex engineering effort which requires, among others, to transfer theoretical results to practice, and to integrate state-of-the-art solutions from a diverse set of domains.

Building a teaching support system is inherently complex and requires the implementation and integration of, among others, modules for providing educational tasks, feedback and advice to students, learning analytics to teachers, interacting with learning management platforms, etc.

There are also challenges specific to teaching support systems for formal foundations, we sketch two of them:

- Algorithms for providing feedback may be known in theory, but may be inefficient in practice and in particular not scale to thousands of users. In our experience, this can often be addressed by employing dedicated SAT solvers and by building custom solutions, e.g., for handling symmetries.

- Helping students to learn how to translate between natural language and formal language — typically a first step when attacking real world problems with formal methods — has been avoided so far in most teaching support systems as it requires to integrate natural language processing. While modern large language models are powerful enough for educational tasks for bridging this natural-formal language gap, a lot of data, engineering, and fine-tuning is required. The main challenge is that a teaching support system should provide correct feedback with very high probability. Figure 7 shows a prototype for an educational task where students can specify propositional variables and their meaning in natural language.

A CS education research challenge

Helping students with a teaching support system requires to understand (among others) students' learning behaviour and motivation, as well as common misconceptions and difficulty-generating factors for formal foundations of computer science.

Didactical aspects of advanced teaching support systems for formal foundations of computer science have mostly been ignored in research. For most concepts taught at university level courses, few didactic foundations have been laid and almost no quantitative and qualitative studies have been done. As a result, there are few guidelines – for example, what common misconceptions students have and how to overcome them, or which factors determine the difficulty of exercises – that can be given to designers of teaching support systems.

Example research questions that have not been addressed for formal foundations of computer science in higher-education contexts are:

(i) How do teaching support systems affect students' learning behaviour and motivation?

(ii) How can teaching support systems be set up to be most effective for heterogeneous groups of students?

(iii) What misconceptions and difficulty-generating factors hinder student success in formal foundations of CS?

(iv) Do (personalized) interventions increase the impact and use of the teaching support system?

Answering these and similar research questions requires a tight integration of expertise in CS education research, educational psychology, formal foundations of computer science, and in building teaching support systems. Data provided by teaching support systems such as Iltis is essential.

Acknowledgments

Many people are contributing to the Iltis project in various ways. Without them, the system would not exist in its current form. The implementation of the system has been supported by Sven Argo, Shrutarv Awasthi, Ariane Blank, Lukas Dienst, Gaetano Geck, Alina Ignatova, Tristan Kneisel, Alexandra Latys, Artur Ljulin, Johannes May, Jan Michalak, Sebastian Peter, Lukas Pradel, Lars Richter, Patrick Roy, Jonas Schmidt, Cedric Siems, Daniel Sonnabend, Fynn Stebel, Felix Tschirbs, Patrick Wieland, Oskar Wilke, and Jan Zumbrink.

The design of the user interface was supported by Christine Orhan. The creation of the course material has been supported by Jill Berg, Tom-Felix Berger, Jan Eyll, Alicia Gayda, Elias Radkte, Mira Schwartz, Dennis Stanglmair, Marco Wojtek, and Cara Volbracht.

Florian Schmalstieg, Thomas Schwentick, Christopher Spinrath, and Nils Vortmeier are providing advice for many aspects of the project.

The example on how to provide extensive feedback for modelling with context-free languages is from joint work of Marko Schmellenkamp and Thomas Zeume with Sven Argo, Cedric Siems, Fynn Stebel, and Sandra Kiefer. The example on how to provide interactive exercises on computational reductions is from joint work of Fabian Vehlken and Thomas Zeume with Julien Grange and Nils Vortmeier. The research questions for CS education research are from a joint project of Thomas Zeume with Philipp Döbler and Jakob Schwerter.

This work is supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grant 448468041.

References

[1] Der Iltis ist das Tier des Jahres 2024. `https://www.pronatura.ch/de/tier-des-jahres-2024-iltis`. Accessed: 2024-02-15.

[2] Andrea L. Beach, Charles Henderson, and Noah Finkelstein. Facilitating change in undergraduate STEM education. *Change: The Magazine of Higher Learning*, 44(6):52–59, 2012.

[3] Loris D'Antoni, Martin Helfrich, Jan Křetínský, Emanuel Ramneantu, and Maximilian Weininger. Automata tutor v3. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification – 32nd International Conference, CAV 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2020.

[4] Gaetano Geck, Artur Ljulin, Sebastian Peter, Jonas Schmidt, Fabian Vehlken, and Thomas Zeume. Introduction to Iltis: an interactive, web-based system for teaching logic. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018*, pages 141–146. ACM, 2018.

[5] Gaetano Geck, Christine Quenkert, Marko Schmellenkamp, Jonas Schmidt, Felix Tschirbs, Fabian Vehlken, and Thomas Zeume. Iltis: Teaching logic in the web. *CoRR*, abs/2105.05763, 2021.

[6] Gesellschaft für Informatik e. V. Empfehlungen für Bachelor- und Master-Programme im Studienfach Informatik an Hochschulen. `https://gi.de`, 2016.

[7] Seymour Ginsburg and Edwin H. Spanier. Bounded algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.

[8] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA, 2013.

[9] Susan R. Singer, Natalie R. Nielsen, and Heidi A. Schweingruber. Discipline-based education research. *Washington, DC: The National Academies*, 2012.

[10] The learning tools interoperability protocol. `https://www.1edtech.org/standards/lti`. Accessed: 2024-02-24.