education sciences

MDPI

Article

Improving Comprehension: Intelligent Tutoring System Explaining the Domain Rules When Students Break Them †

Oleg Sychev *, Nikita Penskoy, Anton Anikin *, Mikhail Denisov and Artem Prokudin

Software Engineering Department, Electronics and Computing Machinery Faculty, Volgograd State Technical University, 400005 Volgograd, Russia; penskoy@vstu.ru (N.P.); denisov@vstu.ru (M.D.); prokudin@vstu.ru (A.P.)

* Correspondence: o_sychev@vstu.ru (O.S.); anton.anikin@vstu.ru (A.A.)

† This paper is an extended version of our paper published in Proceedings of 17th International Conference on Intelligent Tutoring Systems, Virtual Event, 7–11 June 2021.

Abstract: Intelligent tutoring systems have become increasingly common in assisting students but are often aimed at isolated subject-domain tasks without creating a scaffolding system from lower- to higher-level cognitive skills, with low-level skills often neglected. We designed and developed an intelligent tutoring system, CompPrehension, which aims to improve the comprehension level of Bloom's taxonomy. The system features plug-in-based architecture, easily adding new subject domains and learning strategies. It uses formal models and software reasoners to solve the problems and judge the answers, and generates explanatory feedback about the broken domain rules and follow-up questions to stimulate the students' thinking. We developed two subject domain models: an Expressions domain for teaching the expression order of evaluation, and a Control Flow Statements domain for code-tracing tasks. The chief novelty of our research is that the developed models are capable of automatic problem classification, determining the knowledge required to solve them and so the pedagogical conditions to use the problem without human participation. More than 100 undergraduate first-year Computer Science students took part in evaluating the system. The results in both subject domains show medium but statistically significant learning gains after using the system for a few days; students with worse previous knowledge gained more. In the Control Flow Statements domain, the number of completed questions correlates positively with the post-test grades and learning gains. The students' survey showed a slightly positive perception of the system.

Keywords: Bloom's taxonomy; adaptive learning; cognitive learning; constraint-based tutors; intelligent tutoring system

Citation: Sychev, O.; Penskoy, N.; Anikin, A.; Denisov M.; Prokudin A. Improving Comprehension: Intelligent Tutoring System Explaining the Domain Rules When Students Break Them. *Educ. Sci.* **2021**, *11*, 719. https://doi.org/10.3390/educsci11110719

Academic Editor: Radhi Al-Mabuk

Received: 29 September 2021
Accepted: 3 November 2021
Published: 9 November 2021

1. Introduction

Lately, a large number of Intelligent Tutoring Systems (ITS) have been developed for different subject domains. Less effort has been spent on categorizing the learning tasks they have use and the kinds of skills they have developed to provide adequate scaffolding for developing higher-level skills. The popular Bloom's taxonomy [1–5] of educational objectives and its successor models and approaches based on it, such as Bloom's Revised Taxonomy [6] and Bloom's Digital Taxonomy [7] identify six levels of cognitive skills: remembering, comprehending, applying, analyzing, synthesizing, and evaluating activities. These skills, ideally, should be developed in that order because higher-level skills rely on lower-level ones.

1. Knowledge level of Bloom's taxonomy "involves the recall of specifics and universals, the recall of methods and processes, or the recall of a pattern, structure, or setting" [1].
2. Comprehension "refers to a type of understanding or apprehension such that the individual knows what is being communicated and can make use of the material or idea being communicated without necessarily relating it to other material or seeing its fullest implications" [1].

3. Application refers to the "use of abstractions in particular and concrete situations" [1]. Its main difference from comprehension level is that when demonstrating comprehension, the student is hinted about the abstractions (concepts) he should use while the application level requires understanding the concepts involved in a particular task.
4. Analysis represents the "breakdown of a communication into its constituent elements or parts such that the relative hierarchy of ideas is made clear and/or the relations between ideas expressed are made explicit" [1].
5. Synthesis refers to the "putting together of elements and parts so as to form a whole" [1].
6. Evaluation involves "judgments about the value of material and methods for given purposes" [1].

Learning tasks on the first three levels are mostly simple: in them, any sequence of correct steps leads to a solution. The higher-level tasks often require a strategy to reach the objective: not all correct moves will bring the student to the solution. So on the knowledge, comprehension, and application levels, the difference between cognitive (follow the student step-by-step during problem solution) and constraint-based (check snapshot solution states for not breaking the subject-domain constraints) [8] ITS is negligible.

In relation to programming education, they can be interpreted as follows [9].

1. Remembering implies learning the symbols and syntax of programming language to produce code.
2. Comprehending implies understanding the nature of functions and statements of programming language.
3. Applying means the ability to write the code according to the task requirements, and to define the objects and their behavior using programming language.
4. Analyzing is associated with the abilities to find the logical or syntax errors in the program code.
5. Synthesizing implies the ability to implement complex structures in programming language such as modules, classes, functions that interact with each other to solve some initial tasks.
6. Evaluating is related to making choices about how to approach the given problem, what data structures, programming patterns, algorithms to apply for the task solution. To make such a choice, an appropriate level of experience is required.

Usually, the success of the students during programming courses depends strongly on comprehending the programming concepts taught in the introductory courses, so ITS in this area should analyze learners at various levels of cognition of these concepts to examine students' performance during subsequent stages of the courses [10]. Furthermore, while higher-level objectives are often the true goals of education, concentrating on high-level assessments may result in lessening learning gains as some students try to solve high-level problems without adequate knowledge and comprehension of the subject-domain concepts, a problem that limits their progress severely, e.g., students may have difficulties writing code even given an algorithm since they do not make the connection between pseudo-code and programming-language code [11]. One way to guide a student through the domain concepts as learning objectives, control the results, and provide feedback during the learning process is the microlearning approach. It is an activity-oriented approach that provides learning in small parts, including information and interactive activities to practice [12,13].

Students may gain some knowledge by being given simple quizzes, but developing full comprehension and learning to apply rules requires intelligent support. This makes developing ITS aimed at comprehension and application levels an important challenge. The relative simplicity of comprehension-level assessments allows using formal subject-domain models and software reasoners [14,15]. This paves the way to the development of a multi-domain ITS where subject domains can be attached as plug-ins.

Comprehension-level tutoring systems are most useful when students are introduced to new subject domains including large numbers of new concepts. We chose introductory

programming as a meta subject domain for in-depth study. Our system is unique because it is based on a combination of factors. It uses formal subject-domain models, capable of solving subject-domain problems and determining the causes of the errors in students' answers that can be re-used in other educational software. Another important novelty is introducing feedback in the form of questions instead of statements. This helps students by stimulating thinking and information retrieval, while it also allows the system to pinpoint the exact cause of the student's error.

Furthermore, our system offers systematic coverage of all the ways to break subject-domain rules that are used to measure the students' performance and determine the suitable problems for them to solve using a pedagogical strategy based on the Q-learning approach [16,17], though the system design allows the easy integration of other approaches such as Bayesian Knowledge Tracing. This provides our system with the ability to classify the learning problems automatically, which is a step in the direction of generating learning problems using an existing program code without a human in the loop. Our long-term aim is to create a system capable of building problem banks in subject domains automatically, selecting the problems to give students according to the kinds of errors they made previously, and maintaining the dialog with the student (including asking follow-up questions) until mastering the subject-domain concepts and laws.

The rest of this article is organized as follows. In Section 2, we will provide an overview of state-of-the-art works in the introductory programming domain. In Section 3, we will describe the architecture and workflow of the developed system. In Section 4, we will describe the two subject domains developed to evaluate the system. In Section 5, we will describe the evaluation of the developed system. We will follow with a brief conclusion in Section 6.

2. Related Work

Modern Intelligent Programming Tutoring Systems (IPTS) provide a wide range of features for learning-process support in different subject domains, such as programming tasks with feedback, quizzes, execution traces, pseudo-code algorithms, reference material, worked solutions, adaptive features, and many others [18]. Some ITPSs allow generating tasks using templates and subject-domain models [19,20], but most of them use only predefined tasks [11,21–25]. They are aimed at different levels of Bloom's Taxonomy objectives—comprehension [19], application [11,20,21], analysis and synthesis [10,22–26]; some of them include limited support for underlying levels. The systems utilize different question types—single choice [11], multiple choice [11,19,22], drag and drop [22], drop-down choice [22], key in solution [22], and code fragments [11]. The feedback ranges from pass/fail results, errors, and the last correct line [11,13,27] to task-oriented predefined hints [22] and derived hints [20,23]. Most of the systems use hardcoded models (e.g., [19,22]); a few systems, such as [20], use formal models to represent the subject-domain laws.

Reviewing related works, we used the following criteria.

- Domain-independence requires using an open model for representing subject domains, including concepts, their relations, and laws.
- Levels of Bloom's Taxonomy [1,6] (cognitive domain) that IPTS is aimed at. If the high levels are targeted, how much are they supported by developing low levels first?
- Question and task types. They affect learning objectives and the available feedback.
- Feedback type provided.

The results are shown in Table 1.

Table 1. Intelligent programming tutoring systems comparison.

ITS	Open Subject-Domain Model	Rules Modelling	Bloom's Taxonomy Levels	Question Types	Feedback Type	Tasks Generation
Fully automated tutor (A. N. Kumar) [19]	inner model for hypothetical language with Pascal syntax	Inner rules	Compre-hension	multiple-choice	automatic solving the generated problems, on-board answer generation, automatic partial credit, dynamic feedback generation and the ability to explain why an answer is incorrect	templates-based automatic problem generation for 5 pre-defined topics
Intelligent Algebra Tutor (Algebra Notepad) [20]	models valid algebraic operations using answer set programming (ASP)	logic programming language ASP to model if-then production rules	Application, Comprehension	Limited set of actions that students can apply to equations on every step of task solving	Automatically generates step-by-step explanations directly from the ASP model	Single underlying model to generate algebra problems, all valid solutions, and intelli-gent explanations
PyKinetic [22]	Inner model for Python language	Inner rules	Application, Analysis, Synthesis	Multiple choice, Drag and drop, Drop-down choices, Key in solution	Common feedback provided to help the student to fix the error, and task-oriented predefined hints (one hint for each task defined).	Predefined tasks (code fragments and test cases)
Programming Tutor for Haskell [23]	embedded domain-specific language for defining strategies (basic steps to incrementally construct a program)	refinement rules for Haskell functional language, used for reasoning	Synthesis	Works with program code	6 feedback services generate hints derived from the strategy	no
AlgoTutor [11]	internal. C++ language and pseudo-code are supported for algorithmic structures learning and training	internal	Comprehension, Application	Single choice, multiple choice, code fragments	pass/fail result, errors highlighting, last correct line highlighting	no

Most IPTS are built for a single subject domain [18,28]. A common issue is missing levels of Bloom's taxonomy when ITS provides students only high-level learning tasks without providing adequate support for developing lower-level cognitive skills [11], e.g., [23,24] are aimed at the synthesis level, so their feedback provides no or minimal information on the application and comprehension levels. To fully utilize adaptive abilities, ITS needs a large bank of different tasks. Most of the existing systems provide a limited set of predefined tasks [11,22,23,29]. In many of them, this set can be extended by a teacher [22,23,29]. One of the interesting and relevant approaches for adaptive tasks generation and testing is a "notional machine" conception, i.e., an abstract computer for executing programs. This is noted in the literature as a major challenge in introductory programming learning [30,31] because teaching programming paradigms requires using multiple notional machines at different levels of abstraction. This approach covers learning on different levels of Bloom's taxonomy, including higher levels.

Many IPTS use visualization as a powerful means of illustrating the task and its solutions to easier understanding the problem, its possible solutions, and errors [32,33]. Interactive visualization can provide additional feedback during the task solving to facilitate the student's progress [32,34,35]. Textual feedback can be helpful as well [29,36–38].

To uncover the full power of adaptive assessments, ITS needs either a way to generate new learning tasks according to the situation or a large bank of various tasks. Some works advocate problem generation on the formal subject-domain model [19,20], but the generated problems require human verification, limiting their number.

3. CompPrehension: Models, Architecture, and Workflow

We propose a new architecture of a multi-domain intelligent tutoring system on the comprehension level. The main goal of this architecture is the flexibility of the developed system along the four main axes, represented by the plug-in types:

1. Domain plug-ins encapsulate everything related to their subject domains, making the rest of the system domain-independent;
2. Backend plug-ins allow interaction with different software reasoners, solving tasks by using provided laws and facts;
3. Strategy plug-ins assess the level of demonstrated knowledge for the student, and choose the pedagogical interventions and the next question if necessary;
4. Frontend plug-in encapsulates the user interface for teachers and students.

Plug-ins exchange information through the core, but they are independent of each other, making the system's pedagogical parts (strategies) agnostic of the subject-domain knowledge, and vice versa.

The requirement to generate semantic explanations for each possible error, including the broken rules; thus, fostering understanding of the subject domain limits the tutoring system to closed-answer questions because determining the exact reasons for all possible errors in open-answer questions is impossible. Four kinds of questions were chosen for the developed system:

1. Single choice (choosing one element from the given set);
2. Multiple choice (choosing any number of elements from the given set);
3. Match (matching each element from the given set with an element for another set);
4. Order (ordering elements of a given set; elements may be omitted or used several times).

3.1. Comprehension-Level Model of Subject Domain

As is shown in [39], comprehension-level modeling requires axiomatic definitions of all the important properties of the taught concepts. Therefore, while a subject-domain ontology serves as the basis of the model, it is enhanced with rules of two types:

- Positive rules or productions allow inferring the correct answer from the formal problem definition;
- Negative rules or constraints allow catching errors in student's answer.

As in comprehension-level tasks any complete sequence of correct steps always leads to a correct answer, the negative rules can be inferred from positive rules by applying negation, thus giving the system the full set of possible errors; each error has a template for explanation generation. One positive rule can spawn several negative rules, i.e., there may be several ways to break one law. This makes negative rules the best way to measure student's knowledge. However, negative rules can be complex (i.e., there may be several reasons for making the same error in the given situation); in this case, the tutoring system can either give a complex explanation or generate follow-up questions to determine the particular fault reason.

While rules (laws) define all important properties of the concepts, one type of knowledge remains attached to the concepts themselves: the student's ability to identify individual objects of the given concept in the problem definition in its human-readable form. Everything else is taught using the rules.

3.2. Architecture

Figure 1 shows the proposed architecture and the chief responsibilities of its components. Strategy plug-ins analyze the learning situation (a sequence of correct and incorrect applications of subject-domain laws during previous interactions of the student), determine the information to show to the learning and the allowed actions, and form a request for generating the next question or decide that the exercise is complete.

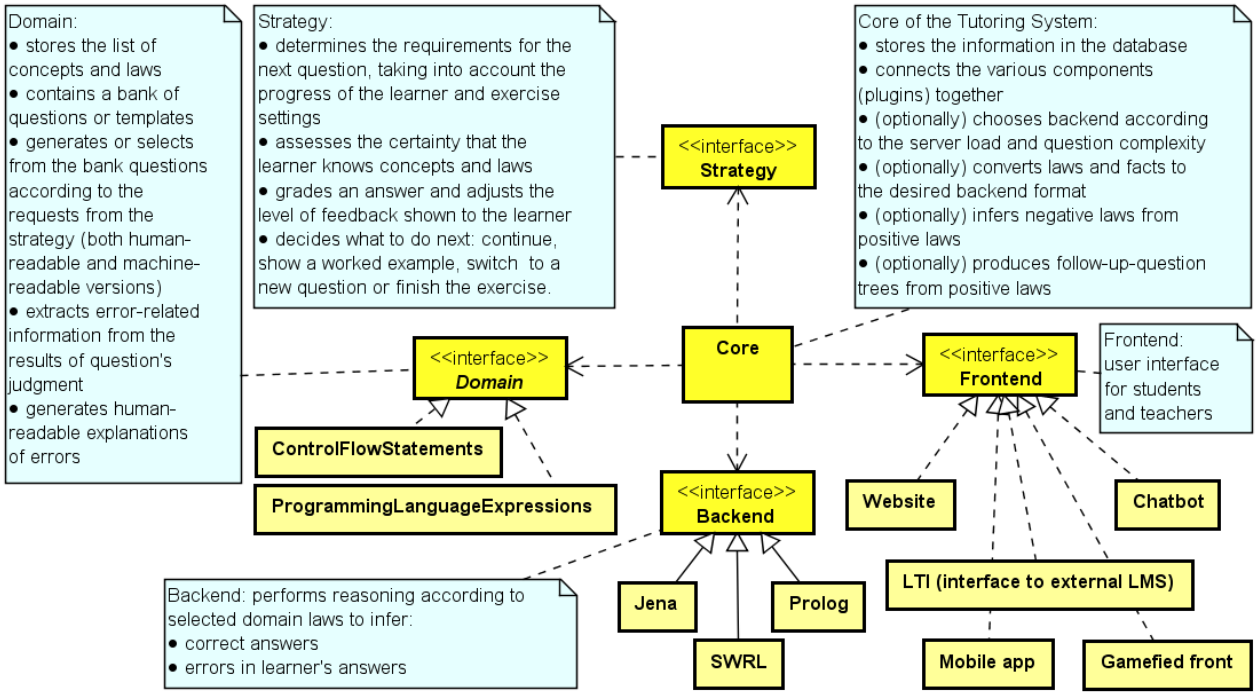

Figure 1. CompPrehension architecture as a UML component diagram.

After conducting experiments measuring wall reasoning time and memory usage for two different subject domains, we found that there is no single reasoner that can outperform all the others. For the Control-Flow Statements domain, the Jena reasoner proved to be the fastest, while for the Expressions domain, Clingo performs better (see Figures 2 and 3). Therefore, we developed backend plug-ins to integrate external reasoners for use by the subject domains.

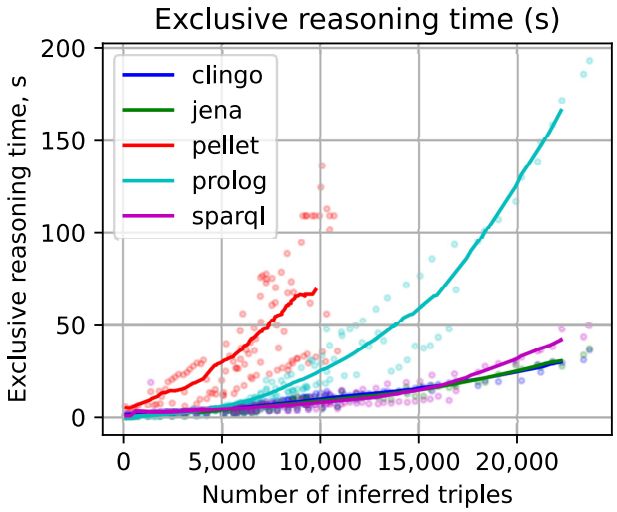

Figure 2. Exclusive reasoning time for the Control Flow Statements domain.

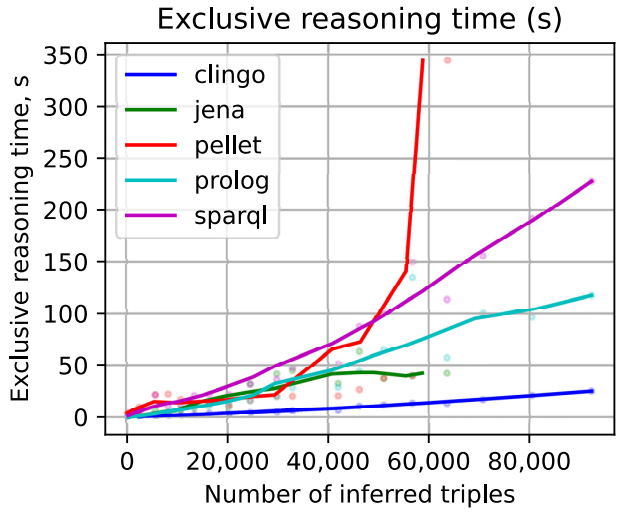

Figure 3. Exclusive reasoning time for the Expressions domain.

Domain plug-ins encapsulate everything related to a particular subject domain, including the formal model (for software reasoner) of the domain (i.e., its concepts and laws) and the particular problem (individuals and facts), and human-readable versions of problem formulations and error explanations. However, subject domains are separated from both pedagogical decisions (which is the responsibility of strategies) and solving the problem according to the formal model (which is done by backends). This allows the efforts required to be minimized, a new subject domain to be added to the system and subject domains with different tutoring models and reasoners to be combined in a versatile way for better efficiency. Another important feature of subject-domain plug-ins is tagging, allowing the system to limit the concepts and laws used in the exercise; tags let a single subject domain cover a group of similar tasks (e.g., the same tasks in different programming languages), re-using the rules.

Frontends encapsulate domain-independent user interface. This allows integration with modern learning management systems through standards such as Learning Tools Interoperability (LTI), using special interfaces for increasing accessibility to particular user categories or providing gamified experience, and using mobile interfaces or even messaging software to interact with the system. Frontend plug-ins can transform complex questions to simpler ones if their user interface is restricted, e.g., a question including

ordering a set of elements can be transformed into a set of single-choice questions "choose the next element".

3.3. Typical Workflow

As domain plug-ins provide rules for finding a correct solution, the system can combine providing worked examples and guiding students through the task-solving process. A typical workflow for task solving is shown in Figure 4 and works as follows:

1. The strategy creates a request for a question based on the exercise settings, student's performance, and difficulty of the laws and concepts.
2. The domain generates a question based on the strategy's request, including machine-solvable and human-readable versions.
3. The backend solves the question, finding the correct answer.
4. The student reads the question and provides a (possibly partial) answer through the frontend.
5. The core transforms the student's answer to backend facts.
6. The backend judges the answer, determining its correctness, errors, and their fault reasons (the sentence).
7. The domain interprets this sentence, transforming backend facts to subject-domain law violations, and generates human-readable feedback (e.g., error explanation).
8. The strategy adjusts the feedback level.
9. The student watches the feedback, possibly requesting more feedback.
10. The strategy chooses to show a worked example for the next step, to ask a follow-up question, to continue with the current question, to generate a new question, or consider the exercise completed (the strategy can also let the student choose among some of these options).

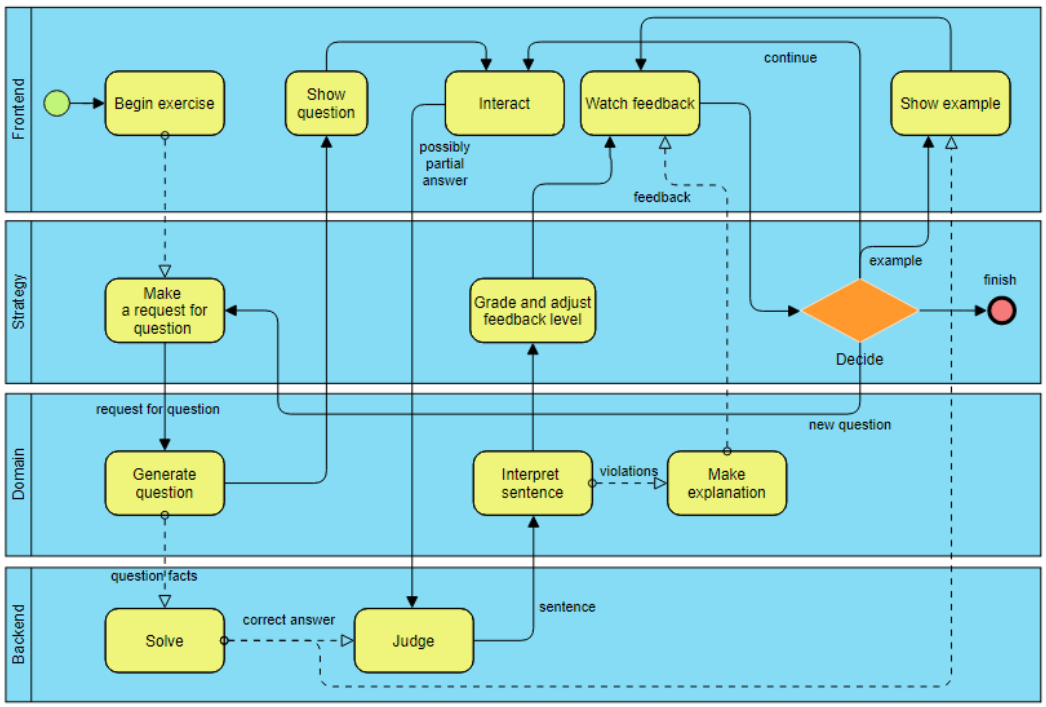

Figure 4. Workflow for CompPrehension system.

4. Developed Domains

As a proof of concept and in order to evaluate the developed system, we developed two domain plug-ins related to teaching programming: Control Flow Statements and Expressions.

4.1. Control Flow Statements

This model is aimed at teaching basic control-flow structures (sequences, alternatives, and loops) using the task of creating a trace for the given algorithm knowing the values of control conditions as they are evaluated. Control Flow is the basic concept of the imperative programming paradigm that makes it an important element in many introductory programming courses. The domain supports the C++, Java, and Python programming languages.

The domain concepts are defined using an ontology. The algorithm is represented as an abstract syntax tree, as in [40]. The classes used for it are shown in Figure 5. The trace is a linked list of Act (i.e., an act of execution) instances connected by the has_next property; it shows the step-by-step process of the algorithm execution (see Figure 6). Single Act instances are created for leaf nodes of the algorithm tree; a pair of corresponding acts (Act_begin and Act_end) represents the boundaries of the other nodes, enclosing the acts of the nested statements. The acts that evaluate control conditions of alternatives and loops contain the values of their conditions. Currently, the domain model contains 29 algorithm elements, 7 trace acts, 29 kinds of errors, 27 explanations for correct acts (total 92 classes), 30 properties, 16 positive rules, 41 negative rules, and 52 helper rules (totaling 99 underlying rules).

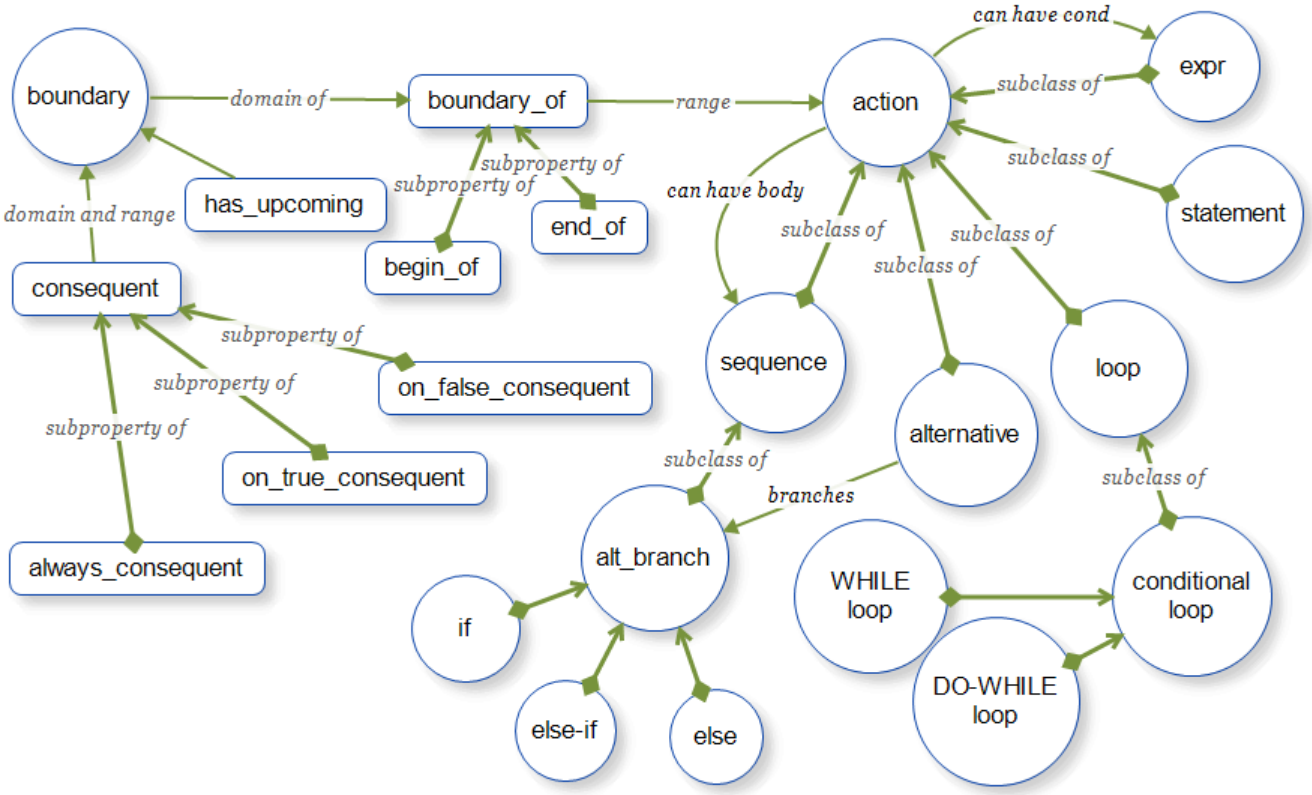

Figure 5. Control Flow Statements domain model: partial hierarchy of concepts for algorithmic structures.

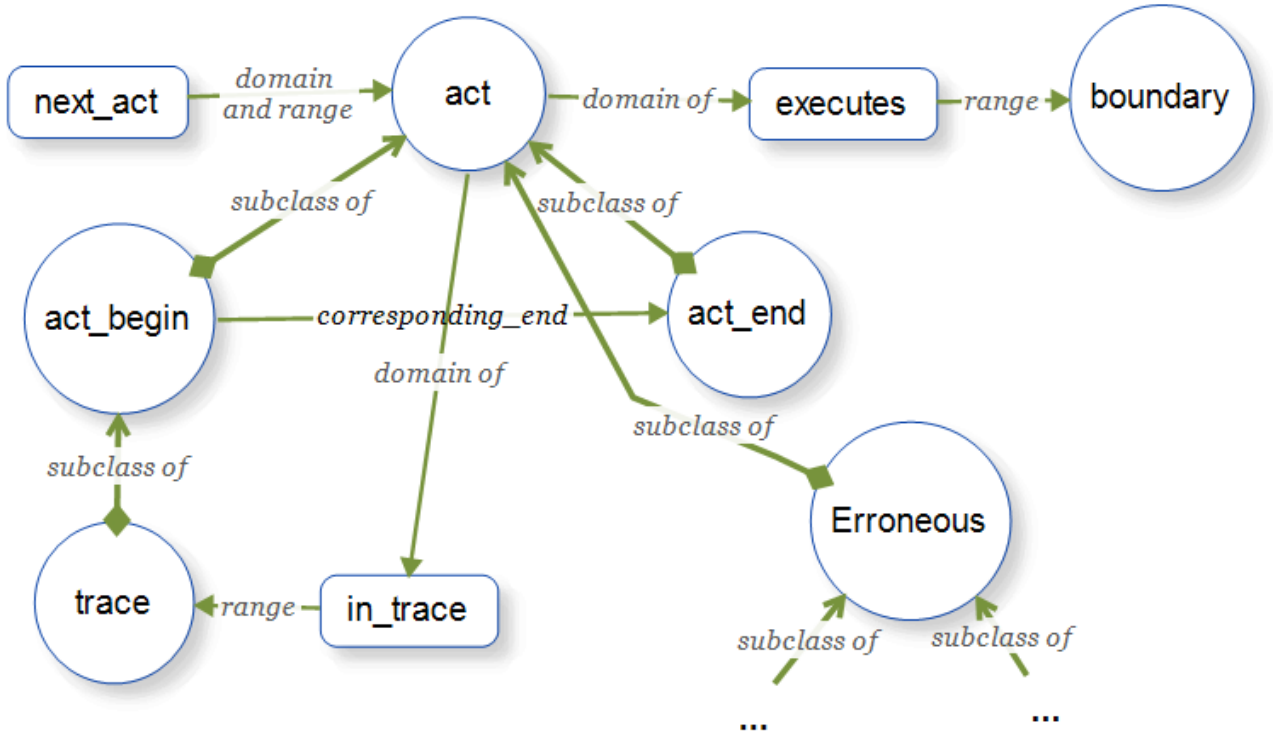

Figure 6. Control Flow Statements domain model: partial hierarchy of concepts for algorithm's trace.

The Control Flow Statements domain uses order tasks, requiring the student to put the acts of execution of the control-flow statements in the correct order; some statements can be executed several times while others may not be executed at all. Figure 7 shows an example of the problem text and a partial answer. The student can use "play" and "stop" buttons embedded in the algorithm to add more acts of execution. If the new Act is correct, it is added to the trace; if it is incorrect, the student receives a message explaining their error and can try another trace element. It is also possible to ask the program to demonstrate a correct step if the student is lost, exploring the example. The ability to demonstrate some steps while leaving the student the others allows our system to narrow down existing questions if the strategy needs to verify or develop the student's knowledge regarding some particular subject-domain law.

Consider the example shown in Figure 7. The student began the trace correctly, but then made an error, starting a loop iteration after its control condition evaluates to false.

The positive rule calculating the correct Act at this point (the end of the loop because the condition is false) is shown in (1).

$$\begin{aligned} &conditional\_loop(L) \ \wedge \ cond(L,C) \ \wedge \ end\_of(loop\_end,L) \\ &\Rightarrow on\_false\_consequent(C,loop\_end) \ \wedge \ NormalLoopEnd(loop\_end) \end{aligned} \tag{1}$$

The negative rule (39th in Table 2) that caught the error and its context (*precursor*) is shown in (2).

$$\begin{aligned} &act\_end(a) \ \wedge \ while\_loop(L) \ \wedge \ cond(L,C) \ \wedge \ executes(a,C) \ \wedge \\ &expr\_value(a,\textbf{false}) \ \wedge \ student\_next(a,b) \ \wedge \ executes(b,S) \ \wedge \\ &body(L,S) \Rightarrow precursor(b,a) \ \wedge \ IterationAfterFailedCondition(b) \end{aligned} \tag{2}$$

Table 2 lists the negative laws for catching errors and provide the examples of explanatory messages for them.

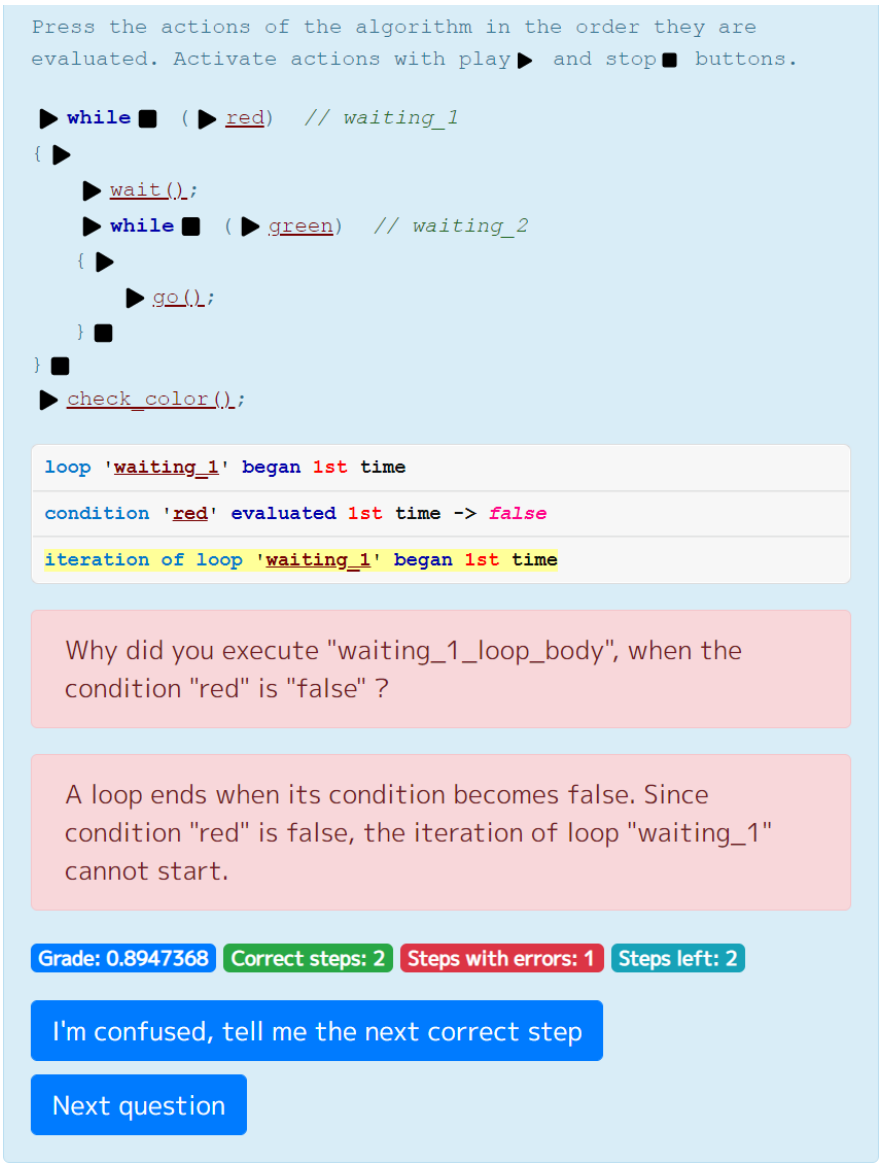

Figure 7. Question example: the algorithm and the partial trace with an error.

4.2. Expressions

This domain is aimed at teaching expression evaluation order using the primary task of choosing the next operator to be evaluated (i.e., ordering question for the operators in the given expression that requires using each operator once and can have several correct orders). Understanding expression evaluation is the basic skill for computer science education; its mastery serves as a basis for developing more complex skills such as analyzing and writing program code. The domain supports the C++ and Python programming languages; the support for other languages can be added using tags for subject-domain laws.

The order of evaluation in modern programming languages is a partial order. Evaluation order is defined by the sequenced-before relation, based on the abstract syntax tree (AST) of the expression. It can be represented as a directed a cyclic graph in which child nodes must be evaluated before their parent nodes.

$$--a + b - c * d \,||\, f(\, e \,\&\&\, j\,,\, k \,/\, m\,) \tag{3}$$

Table 2. Negative laws for catching mistakes and their messages in the Control Flow Statements domain.

	Mistake Name	Message Example
1	UpcomingNeighbour	Why did you skip loop "waiting", action "get_ready()" ?
2	NotNeighbour	Why did you execute branch "if-green" ?
3	WrongCondNeighbour	Why did you execute branch "if-green", when the condition "green" is false ?
4	BeginEndMismatch	Bad trace: loop "waiting" ended as selection "choose".
5	EndedDeeper	An action ends only when all its nested actions have ended, so A cannot end until K ends as K is a part of A.
6	EndedShallower	Selection "choose" cannot end loop "waiting" as loop "waiting" contains selection "choose".
7	WrongContext	A cannot be executed inside of B because A is not a direct part of B.
8	OneLevelShallower	A cannot be executed within C because A is an element of P, so start P first.
9	TooEarlyInSequence	A sequence executes its nested actions in order, so B cannot be executed before A.
10	TooLateInSequence	A sequence executes its nested actions in order, so A cannot be executed after B.
11	SequenceFinishedTooEarly	A sequence always executes all its actions. The sequence A cannot finish until actions: X, Y, Z are executed.
12	SequenceFinishedNotInOrder	Sequence "if-ready" cannot end until it starts.
13	DuplicateOfAct (of sequence)	A sequence executes each its action once, so each execution of P can contain only one execution of X.
14	NoFirstCondition	Selection statement "choose" should start with evaluating its first condition "red".
15	BranchNotNextToCondition	Selection statement "choose" can execute the branch "if-red" right after condition "red" only.
16	ElseBranchNotNextToLastCondition	Selection statement "choose" cannot execute the branch "ELSE" until its condition "green" is evaluated.
17	ElseBranchAfterTrueCondition	Selection statement "choose" must not execute its branch "ELSE" since condition "green" is true.
18	CondtionNotNextToPrevCondition	Selection statement "choose" can evaluate its condition "green" right after the condition "red" only, if "red" is false.
19	ConditionTooEarly	Selection statement "choose" cannot evaluate its condition "green" until the condition "red" is evaluated.
20	ConditionTooLate	Selection statement "choose" should evaluate its condition "green" earlier, right after condition "red" is evaluated.
21	ConditionAfterBranch	Selection statement "choose" must not evaluate its condition "green" because the branch "if-red" was executed.
22	DuplicateOfCondition	Selection statement "choose" must not evaluate its condition "red" twice.
23	NoNextCondition	A selection statement evaluates its conditions in order up to the first true condition. Selection statement "choose" should evaluate its condition "green" next because the condition "red" is false.
24	BranchOfFalseCondition	A selection statement executes its branch only if its condition is true. Selection statement "choose" must not execute the branch "if-green" because its condition "green" is false.
25	AnotherExtraBranch	A selection statement executes only one branch. Selection statement "choose" must not start its branch "else" because the branch "if-red" was executed.
26	BranchWithoutCondition	A selection statement executes its branch when the branch condition evaluates to true. Selection statement "choose" must not execute the "if-red" without evaluating its condition "red" first.
27	NoBranchWhenConditionIsTrue	A selection statement executes its branch when the corresponding condition is true. Selection statement "choose" must execute the branch "if-red" because its condition "red" is true.
28	LastFalseNoEnd	When all conditions of a selection statement are false and "ELSE" branch does not exist, the selection does nothing. Selection statement "choose" does not have an "else" branch so it must finish because its condition "green" is false.
29	AlternativeEndAfterTrueCondition	When a condition of a selection statement evaluates to true, the selection executes the corresponding branch. Selection statement "choose" should not finish until the branch of successful condition "red" is executed.
30	NoAlternativeEndAfterBranch	A selection statement finishes after executing one branch. Selection statement "choose" executed its branch "if-green" and should finish.
31	LastConditionIsFalseButNoElse	A selection statement executes its "ELSE" branch only if all conditions are false. Selection statement "choose" must execute its branch "ELSE" because the condition "green" evaluated to false.
32	NoIterationAfter-SuccessfulCondition	A WHILE loop continues if its condition is true: its new iteration must begin. A new iteration of the loop "waiting" must begin because its condition "ready" is true.
33	LoopEndAfterSuccessfulCondition	A WHILE loop continues if its condition is true: its new iteration must begin. Its too early to finish the loop "waiting" because its condition "ready" is true.
34	NoLoopEndAfterFailedCondition	A WHILE loop ends when its condition becomes false. As the condition "ready" is false, the loop "waiting" must end.
35	LoopEndsWithoutCondition	Since the condition "ready" is not evaluated yet, the loop "waiting" must not end.
36	LoopStartIsNotCondition	A WHILE loop is a pre-test loop. So the loop "waiting" should start by evaluating its condition "ready".
37	LoopStartIsNotIteration	A DO loop is a post-test loop. Therefore, loop "running" should begin with an iteration.
38	LoopContinuedAfterFailedCondition	A loop ends when its condition becomes false. Since condition "ready" is false, loop "running" cannot continue.
39	IterationAfterFailedCondition	A loop ends when its condition becomes false. Since condition "ready" is false, the iteration cannot start.
40	NoConditionAfterIteration	After an iteration of DO-WHILE loop, its condition must be evaluated to determine whether to continue the loop or finish it. After an iteration of loop "running", its condition "ready" should be evaluated.
41	NoConditionBetweenIterations	After an iteration of DO-WHILE loop, it is to determine whether the loop continues or ends. Before proceeding to the next iteration of loop "running", its condition "ready" should be evaluated.

Figure 8 demonstrates the evaluation order graph for the expression 3 in the C++ programming language; the purple arrow pointing between the rectangles shows that the left operand of || (logical or operator) should be evaluated before any operator from its right operand. This notation is used to avoid drawing too many lines between the operators in the left and right rectangles.

The ontology rules we developed determine the sequenced-before relations between the expression operators (positive rules), compare it with the student's evaluation order to determine inconsistencies and determine the error by matching (negative rules). The domain code written in Java can then generate error messages from the natural-language templates associated with negative rules.

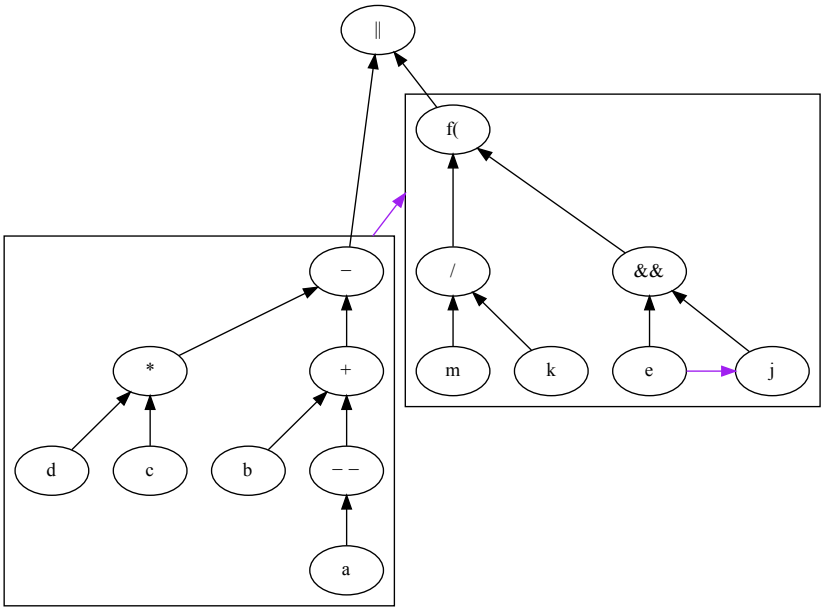

Figure 8. Expression evaluation order as a graph.

The formal reasoning model receives the expression split into tokens. For each token, its position in the expression and the text of the token are set. An abstract syntax tree for the given expression is built after it. AST construction is an iterative process: at each step, an operator whose operands are already evaluated is found, and its operands are determined. In order to make the model compatible with the reasoners which cannot change or delete existing facts—only add new facts—we needed special measures.

We copied the tokens to create a separate layer for every step of AST construction so that we could assign different labels to the same token, storing its current state, at different steps. A unique data property "step" is set for the tokens in each layer. This is performed sequentially: after a new operator and its operands are found, all the specified properties of the current layer are copied to the next one, except the auxiliary properties that do not make sense at the next iteration (such as the property showing the operator with the highest priority) and the properties for token labels that change between the steps.

There are three token states represented by labels: initialized, evaluated, and applied. The initialized state is used for operator tokens that are not evaluated yet. Evaluated label is used for the tokens which have already been evaluated during previous steps, but have not been used as operands for other evaluated operators. Operand tokens (variables and literals) start in this state. The applied label is used for the tokens which are already marked as direct operands of some evaluated operator. At the end of the AST building, only one token remains in the evaluated state—the root operator of the syntax tree; all other tokens should be in the applied state.

The built AST serves as the basis for the graph of sequenced-before relations between the operators. Some operators require a strict order of evaluation for their operands; this leads to adding more edges to the graph. For example, in the C++ programming language, the logical operators and the comma operator require a full evaluation of their left operand before the evaluation of any operator belonging to their right operand; the ternary operator also evaluates its first operand before the others. After taking into account these relationships, the full graph of the operators' dependencies, encompassing every possible correct answer to the question, is built.

A student should click on the operators in the expression in the order they are evaluated. The student's answer is passed to the reasoner as a sequence of operator evaluation. If the order of a pair of operators linked by a sequenced-before graph edge contradicts the student's answer (i.e., the parent is evaluated before the child), a student's error is detected. Using the information from the edge nodes and the type of the violated edge, the negative

rules classify the error and link all the necessary nodes to it (the errors concerning the strict operand order need to have the operator that causes the strict-order relationship linked to them to generate the full message). This information is used to store the student's error in the database (to evaluate the student's knowledge and better choose the next question) and generate the error message to show to the student. In Figure 9, you can see the error messages for an attempt to evaluate the addition operator at position 4 broke two subject-domain rules: the first is related to operator precedence, the second to operator associativity.

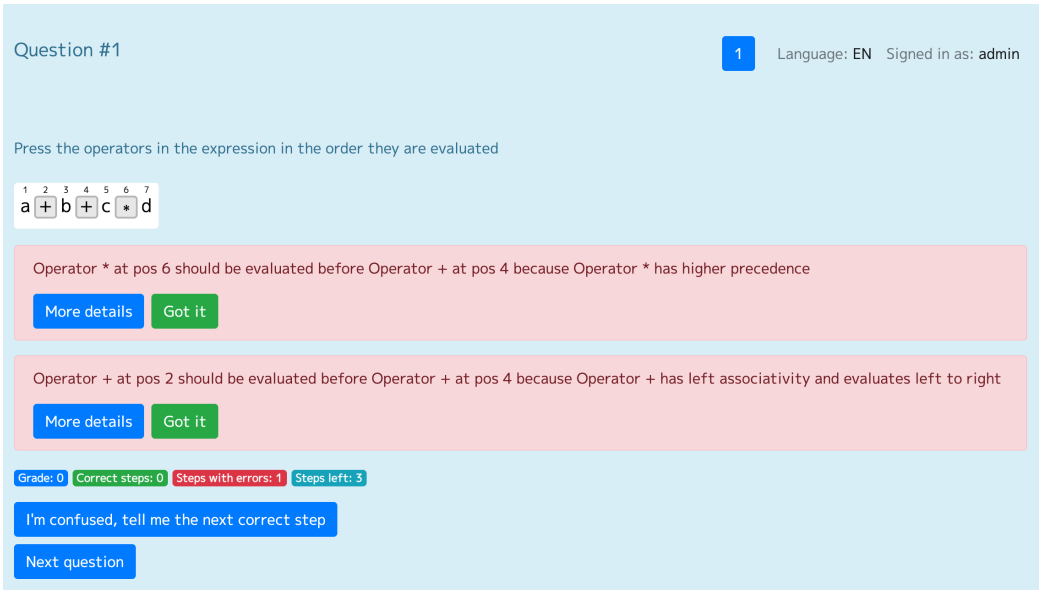

Figure 9. Expression evaluation domain: error messages.

The list of all possible basic errors is shown in Table 3.

Table 3. Negative laws for catching errors and their messages in the Expressions domain.

Error Name	Message Example
HigherPrecedenceRight	Operator * at pos 4 should be evaluated before Operator + at pos 2 because Operator * has higher precedence
HigherPrecedenceLeft	Operator + at pos 4 should be evaluated before Operator < at pos 6 because Operator + has higher precedence
LeftAssociativityLeft	Operator + at pos 2 should be evaluated before Operator − at pos 4 because Operator + has the same precedence and left associativity and evaluates left to right
RightAssociativityRight	Operator = at pos 6 should be evaluated before Operator += at pos 4 because Operator = has the same precedence and right associativity and evaluates right to left
InComplex	Operator + at pos 2 should be evaluated before Operator * at pos 5 because expression in parenthesis is evaluated before the operators outside of them
	Operator/ at pos 4 should be evaluated before parenthesis (at pos 2 because function arguments are evaluated before function call
StrictOperandsOrder	Operator< at pos 2 should be evaluated before Operator > at pos 6 because the left operand of the Operator || at pos 4 must be evaluated before its right operand

However, these error messages and situations are complex, and they may not be easily understandable for all students. Furthermore, there are many possible causes for the same

error. Consider the situation in Figure 9. The student thinks that the addition operator at position 4 can be evaluated, but why? There are many possible causes for it, for example:

- The student does not know that addition is left-associative;
- The student does not know that precedence must be checked before associativity;
- The student does not know the relative precedence of addition and multiplication.

To learn the true cause of the student's error and stimulate thinking about the problem, the domain can ask the student a series of small multiple-choice questions related to the error made—follow-up questions that the student can obtain by pressing on the "More details" button or if the strategy plug-in decides that the student should answer these questions.

Follow-up questions are shown in a subsection of the specific error (see Figure 10). The messages about the correctness of the answers to follow-up questions are shown for several seconds below the question; it is also emphasized by the message color (green for correct answers, red for wrong answers). After that, the next question or the final message is shown.

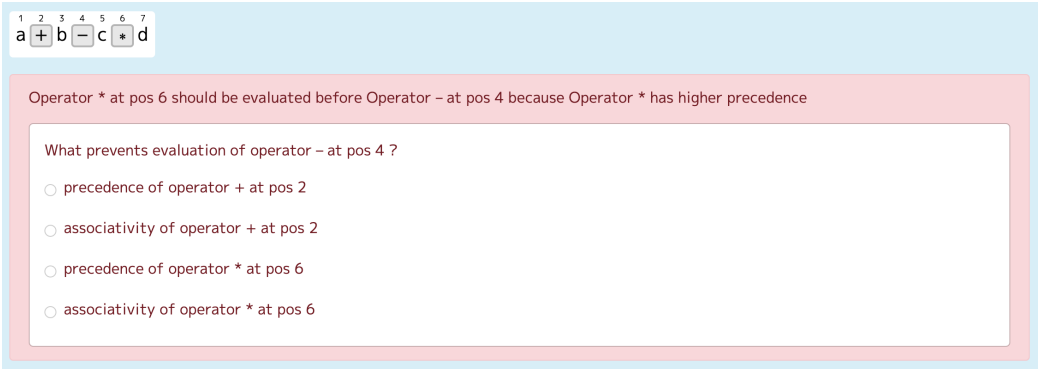

Figure 10. Expressions domain: an example of a follow-up question.

Consider the following example of follow-up question for the situation in Figure 9.

- Program: What prevents evaluation of operator + at pos 4?
- Student: Precedence of operator * at pos 6.
- Program: Correct.
- Program: Which operator has highest precedence: + or * ?
- Student: Operator +.
- Program: Wrong.
- Program: No, operator * has precedence over operator +, so the operator * at pos 6 should be evaluated before the operator + at pos 4.

In this case, the dialog lets the program determine the exact subject-domain fact that the student got wrong and give a detailed explanation. Sometimes, however, the student can understand their error while answering the follow-up questions and the resulting message must be affirmative. Consider the example shown in Figure 10. The dialog between the student and the system will look as follows.

- Program: What prevents evaluation of operator − at pos 4?
- Student: Precedence of operator + at pos 2.
- Program: Wrong.
- Program: What influences evaluation order first: precedence or associativity?
- Student: Precedence.
- Program: Correct.
- Program: Which operator has highest precedence: − or + ?
- Student: The same precedence.
- Program: Correct.
- Program: What influences evaluation order of operators with the same precedence?
- Student: Associativity.

- Program: Correct.
- Program: What is the associativity of operator −?
- Student: Left.
- Program: Correct.
- Program: Yes, the operators + and − have the same precedence and left associativity, so the operator + at pos 2 should be evaluated first as it stands to the left of the operator − at pos 4.

Follow-up questions are defined as a finite-state machine in the domain code. In Figure 11, you can see a part of the graph of follow-up questions in the Expressions domain—the questions regarding operator precedence and associativity.

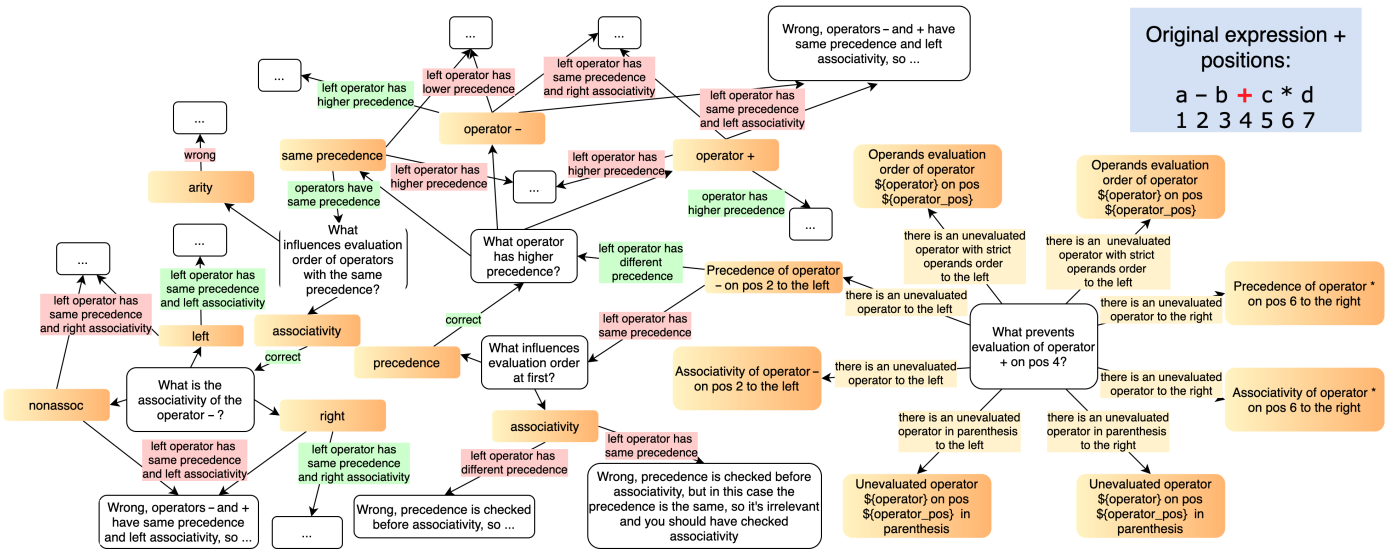

Figure 11. Partial follow-up questions diagram: operator precedence and associativity.

Currently, the domain model contains 75 operator definitions (for the C++ and Python programming languages), 28 logical rules (6 positive, 6 negative, and 16 helper rules), and 35 follow-up questions.

5. Evaluation

First-year undergraduate Computer Science students of Volgograd State Technical University participated in the evaluation of our system. They were asked to complete four steps:

1. Pass a pre-test to measure their baseline knowledge (maximum grade is 10);
2. Work with the CompPrehension system to learn more, preferably until the system tells that the exercise is complete;
3. Pass a post-test to measure their learning gains (maximum grade is 10);
4. Participate in a survey about the system they used.

The acquired knowledge was related to the topics in the "Informatics" (CS0) course they were currently taking and could improve their grades which motivated them for participation. They were given one week to complete everything. The survey was anonymous so that the students could express their opinions freely. Out of the 151 students who attempted the first pre-test (for the expressions domain), 88 students completed all the tasks. Table 4 shows in detail how many students completed various tasks. The smaller number of students attempting the Control Flow Statements domain can be explained by its higher complexity and the effort required to perform the exercise as a typical algorithm trace contains significantly more elements than a typical expression. Another possible explanation is that the Control Flow Statements domain was second in their online course, so some students dropped after completing exercises for the first domain.

Table 4. The number of students who completed tasks.

Domain	Pre-Test	Post-Test	Survey
Expressions	151	135	88
Control Flow Statements	142	129	

5.1. Expressions Domain

The Expressions domain relied partially on the students' previous knowledge—most of them should be acquainted at least with the concept of operator precedence in school. So relatively high grades of the pre-test were expected, and these expectations were confirmed by the students solving more than 50% of the pre-test on average.

The maximum grade for the pre-test and post-test was 10. As Table 5 shows, their average learning gains were relatively small, but, combined with the previous knowledge, it let them master more than 2/3 of the subject-domain knowledge after a short exercise. The paired two-tailed *t*-test showed that the learning gains are statistically significant. The increase in standard deviation showed that the students did not learn uniformly: some of them benefited from using the system more than others.

Table 5. Absolute learning gains for the Expressions domain.

	Pre-Test	Post-Test	Gains	Significance
Avg.	5.62	7.17	1.6	$p = 3 \times 10^{-12}$
Std. dev.	1.9	2.48		

Most of the students completed the pre-test between 5 and 6 grade (out of 10) as shown in Figure 12. We divided the students into two groups by their previous knowledge: students with low pre-test grades (less than 6) and students with high pre-test grades (more than 6). Their learning gains were different (see Table 6: the students with less initial knowledge learned much more. The independent two-tailed *t*-test showed that this difference is statistically significant ($p = 0.001$). This confirmed the hypothesis that developing comprehension of concepts helps less-performing students more. One possible argument against this is that the students with poor initial knowledge had more to gain than the students with good initial knowledge. In order to assess this, we calculated relative learning gains by dividing the absolute learning gains of each student by their maximum possible gains; the difference between the students with low and high pre-test grades is still statistically significant for relative gains even though with much bigger $p = 0.03$.

Table 6. Absolute learning gains for the students with low and high pre-test grades in the Expressions domain.

	Low Pre-Test	High Pre-Test	Significance
Number of students	86	57	
Avg.	2.14	0.77	$p = 0.001$
Std. dev.	2.69	1.92	

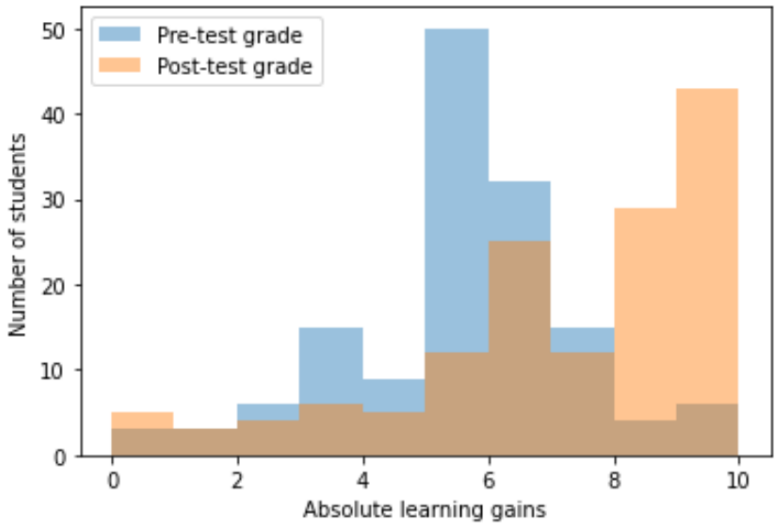

Figure 12. Histogram of the pre-test grades and post-test grades in the Expressions domain.

Table 7 shows the statistics on how the students used the Expressions domain in the system: the number of questions generated for the student, the number of steps to solve the questions the student did, the percentage of the correct steps, the number of hinted steps, and the number of follow-up questions series. Zeroes in the minimum steps show that some students did not try to solve CompPrehension's questions; they only asked for hinting correct steps. The students could also omit some of the questions. On average, a student received 14 questions but fully solved only about five of them, making 36 solution steps (each step required one mouse click). The exercise usage was during homework, and the students were given the possibility to move to the next question without solving the previous question fully, which led to a rather low percentage of completed questions. The average percent of correct steps was more than 80%, but in difficult situations, the students asked for correct solution hints. The low usage of follow-up questions is understandable because this topic is easy for college-level students and so the error explanations were mostly sufficient. The average number of steps (clicks) to solve a question was 2.34; the average percent of hinted steps is 11%. We found no significant correlation between these variables and learning gains.

Table 7. Usage statistics for the Expressions domain.

	Min	Max	Average	Std. Dev.
Total questions	1	136	13.98	18.44
Completed questions	1	14	4.71	2.59
Total steps	0	268	35.84	44.85
Steps per question	0	5	2.34	1.37
Percent of the correct steps	0	100%	81.63%	16.31
Hinted steps	1	57	4.04	8.98
Follow-up questions usages	0	10	0.69	1.45

Generally, while the students' average learning gains of 1.6 out of 10 do not look impressive, they are significant for an exercise that required 36 mouse clicks on average.

5.2. Control Flow Statements Domain

Fewer students chose to participate in the evaluation of the Control Flow Statements domain than in the evaluation of the Expressions domain, possibly because it took more effort. Still, the percentage of the students who finished exercises and the post-test—90%—was almost the same as for the easier Expressions domain. Out of 129

students who attempted the post-test, only 13 students finished their exercise attempts in the system; two of them finished two attempts. This is understandable as there were a lot of laws to learn in this domain, and each question in the attempt required more work, while the system was conservative and only considered the exercise attempt complete when the student demonstrated knowledge of all the subject-domain laws.

The learning gains after using the CompPrehension system were about the same as for the Expressions domain with a slightly lower knowledge level which can be explained by the complexity of this domain. The *t*-test (paired, two-tailed) showed the statistical significance of absolute learning gains (see Table 8). You can see the histogram of the pre-test and post-test grades in Figure 13. It shows that after working with our system, the students were divided into two distinct groups: the students with poor knowledge (the grade is less than 4) and the students with good knowledge (the grade is 6 or more).

Table 8. Absolute learning gains for the Control Flow Statements domain.

	Pre-Test	Post-Test	Gains	Significance
Avg.	4.95	6.73	1.78	$p = 1.6 \times 10^{-16}$
Std. dev.	2.38	2.87	1.96	

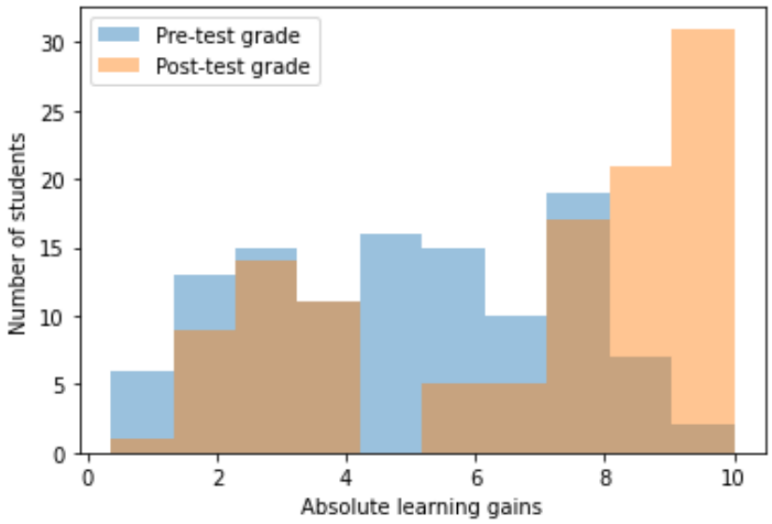

Figure 13. Histogram of the pre-test grades and post-test grades in the Control Flow Statements domain.

As for the Expression domain, the students with lower pre-test grades (less than 6 out of 10) gained more than the students with high pre-test grades. This difference (see Table 9) is statistically significant for absolute learning gains (independent two-tailed *t*-test $p = 0.009$) but is not significant for relative learning gains ($p = 0.06$) by a small margin. This supports the hypothesis that poorly performing students gain more by using comprehension-level intelligent tutoring systems.

Table 9. Absolute learning gains for the students with low and high pre-test grades in the Control Flow Statements domain.

	Low Pre-Test	High Pre-Test	Significance
Number of students	84	44	
Avg.	2.12	1.16	$p = 0.009$
Std. dev.	2.22	1.31	

Usage statistics for the Control Flow Statements domain are shown in Table 10. The students solved about half of the questions they solved in the Expressions domain but per-

formed double the number of solution steps, which is understandable because the Control Flow Statements domain has longer questions. In this domain, the students completed most of the questions they started compared to the Expressions domain that shows their interest and engagement. The percent of correct steps is lower than for the Expressions domain—a more complex subject domain made students make errors more often. However, the increase in using the correct steps hint is not large, and the percentage of using hints to all solution steps decreased significantly, which shows that the error explanation messages were useful most of the time. We did not use follow-up questions in the Control Flow Statements domain for this evaluation because in this domain error explanations are simpler than in the Expressions domain: in the Control Flow Statements domain, each situation has only one next correct step.

Table 10. Usage statistics for the Control Flow Statements domain.

	Min	Max	Average	Std. Dev.
Total questions	1	52	6.59	9.05
Completed questions	1	27	5.48	6.13
Total steps	0	540	64.51	98.7
Steps per question	0	32	7.31	6.80
Percent of the correct steps	0	100%	68.93%	21.4
Hinted steps	0	31	4.67	7.76

The values of the Pearson's correlation coefficient between different variables for the Control Flow Statements domain are shown in Table 11. Weak negative correlation between the percent of errors (failed interactions) and the pre-test grade was trivial, showing that better-prepared students made fewer errors. More interesting are the correlations for the post-test grades: there was a medium positive correlation between the number of completed questions and the resulting knowledge, and a medium negative correlation between the percent of errors and the resulting knowledge. The number of completed questions shows a consistent influence on the learning gains, with a weak positive correlation to absolute learning gains and a medium positive correlation to relative learning gains. This shows that the students who did not hurry to the next question, completing each question, gained more knowledge.

Table 11. Correlation coefficient for the Control Flow Statements domain variables.

	Pre-Test Grade	Post-Test Grade	Absolute Learn. Gains	Relative Learn. Gains
Completed questions	−0.017	0.158	0.138	0.123
Total questions	−0.103	−0.098	0.023	−0.059
Percentage of completed questions	0.139	**0.567**	0.285	**0.433**
Failed interactions	−0.283	−0.239	0.094	0.026
Correct interactions	−0.083	0.106	0.164	0.151
Total interactions	−0.130	0.043	0.161	0.135
Percentage of failed interactions	**−0.369**	**−0.518**	−0.029	−0.271
Hinted steps	−0.290	−0.182	0.146	−0.065

5.3. Survey

Out of the students who participated in the evaluation, only 88 students completed the final survey. The students were asked eight questions using a Likert scale (1—strongly disagree, 5—strongly agree). We used both positive and negative statements about the system to avoid bias in question formulation. The results are shown in Table 12, while most average results are close to neutral, they are consistently above three for the positive

statements and below three for the negative statements that shows moderate support of the developed system. The main problems the students noted in the free text part of the survey were problems of waiting for server response (the server did not bear the load of more than 100 students well during the first days of the evaluation) and the small amount of theoretical material supporting the exercises.

Table 12. Survey results.

Question	Average	Std. Dev.
The interface was easy to understand and use	3.86	1.07
The exercise formulations were clearly stated	3.25	1.22
Some of the questions concerned the problems which were difficult to learn without this application	3.15	1.20
The tasks were repetitive and tiring	2.50	1.18
The explanations helped to solve the exercises	3.48	1.18
The explanations helped to understand the error and avoid it in the future	3.63	1.03
The exercises take too much time	2.97	1.37
I want to use this application to learn more about programming	3.57	1.17

6. Discussion

In this article, we described an intelligent tutoring system aimed at developing comprehension of subject-domain concepts in programming and other well-formalized domains. Some of the similar systems (e.g., [41]) even working in well-formalized domains such as mathematics and programming allow to evaluate and provide the feedback only for the errors in the answer to the final task, or some of the steps of solving it which makes the feedback more general and decreases its quality. This also limits the system's ability to determine the exact cause of the student error and so update the model of the student's knowledge as several kinds of errors can lead to the same wrong answer step.

We enhance this approach by introducing advanced feedback in the form of follow-up questions based on the Socratic questions approach [42,43]. It requires active student participation, thus stimulating information retrieval and thinking in the subject-domain terms, and allowing the system to determine the exact cause of the error the student made. Using questions in addition to statements (a common form of providing feedback in ITS) advances the field and opens the way for new research, determining when and where they are more effective than the regular feedback.

A common approach to generating feedback (see [44]) is based on classifying it on the levels of details (e.g., abstract, concrete, and bottom-out levels) but it often makes feedback content non-systematic: sometimes it is aimed at explaining to the student why they are wrong, while in other cases it just tells the student what to do to proceed to the correct answer without developing comprehension. Both kinds of feedback can co-exist in the same system, e.g., [45]. Our approach advances the field by concentrating on feedback about the subject-domain rules and facts that the student violated when making an error and using this kind of violation to measure the students' knowledge. Our formal models deliberately avoid comparing a student's answer with the correct answer, making supporting domains with many possible correct answers easy.

The same data about the causes of the students' errors are used both for providing feedback and choosing the next question to support learning, unlike [11,22,23], and most of the other solutions reviewed. To estimate the student's knowledge of the subject domain, a model based on the Q-learning approach is used, but other models such as Bayesian Knowledge Tracing can be added easily using the interface for strategy plug-ins. This allows using our approach in guided learning [46,47] and export these data for reviewing by teachers and tutors and to foster integration with Learning Management Systems and Intelligent Textbooks [48] using common domain models.

While developing the described ITS, we faced the following challenges:

1. decoupling pedagogical strategies from subject-domain models;
2. decoupling subject-domain-specific user interface from general question interface;
3. systematic coverage of all the ways to violate subject-domain rules by negative laws;
4. developing a question base to satisfy all possible requests for generating questions.

To make plug-ins implementing pedagogical strategies agnostic of the subject domains, we needed a way to define the knowledge space for a subject domain, i.e., all kinds of facts, rules, and laws students should learn (develop mastery of). We found negative laws, i.e., the ways to make an error, the best measure of the knowledge required to solve a specific question because they are more detailed than the rules for making correct steps (positive laws). The knowledge space also includes concepts, but they are used to measure the student's ability to recognize the objects of that concept in the problem formulation; everything else is modeled using rules. Using this knowledge space, a strategy can measure the student's knowledge and mastery and formulate requests for the new questions based on the kinds of errors that could be done in them.

The user interface subsystem faced a similar problem because many subject domains benefit from domain-specific interfaces. The main challenge there was making as few restrictions on the possible question formulations as possible while placing the controls to answer right into the formulation (c.f., the examples in the Control Flow domain above: while it is possible to offer the student to select the next statement to perform using the regular interface for multiple-choice questions with round buttons below the algorithm, it is highly unfeasible because of the number of possible choices that all have corresponding parts in the question formulation). Our solution was passing the question formulation from the domain plug-in as HTML code, supplemented with an array of "answer objects"—i.e., the objects from which the answer is built—containing button identifiers for the buttons generated by the domain plug-in. This way, the system core, user interface, and strategy can keep track of the student's answer without knowing how it is represented in the question formulation.

Other challenges for the development of this system were systematic coverage of the entire error space and developing a question bank to satisfy the requests for questions generated by the strategy using the given knowledge space. They required a lot of work from the team (though this work is necessary only once per subject domain). We are going to attend to this problem in further research. To cover the error space systematically, we need a formalized representation of the taught intellectual skill that can be done using decisions trees. The question bank can be significantly expanded by mining the question formulations from the existing program code. The key feature of our subject-domain models allowing it is that they can automatically determine the list of the possible errors for a given task, i.e., there is no need for a human question author, specifying which question templates correspond to which topics within the subject domain. This allows large-scale question generation without teacher intervention.

7. Conclusions and Future Work

We designed a domain-independent comprehension-level ITS CompPrehension and developed a working prototype, including two subject domains (Control Flow Statements and Expressions), two backends (SWRL and Jena rules), the LTI Frontend, and a basic strategy. The domains contain tag sets for the C++ and Python programming languages.

The system is capable of selecting a problem from the problem base according to the student's performance, solving it using software reasoning, grading answers, determining fault reasons, showing explanatory feedback, asking follow-up questions, and showing worked examples. The main difference of our system from most of the other intelligent programming tutoring systems is consistently displaying the information about the broken subject-domain rules in the error messages. Integrating a new subject domain requires developing its domain plug-in. As all the questions in the developed system are automati-

cally linked with the concepts and subject-domain laws that should be known for a correct solution, they can be easily integrated with Intelligent Textbooks [48].

The system's evaluation showed that for the both domains, doing exercises produced moderate learning gains that were greater for the low-performing students. Since the students did not receive any additional instructions on the topic during the evaluation process, these learning gains can be attributed only to the developed system. This is in line with the findings of Kumar [44] for code-tracing problems. For exercises in the Control Flow Statements domain, the percentage of completed questions was positively correlated with the post-test grades and learning gains while not being correlated with the pre-test grade. This shows influence over how much the students learned; it may be beneficial to disable the ability to start a new question without completing the previous question. The percentage of errors was negatively correlated with both the pre-test and post-test grades but not with learning gains. This only shows that better-performing students make fewer errors.

The developed system exposes the properties of the subject-domain concepts through simple questions, verifying and developing their comprehension for students. This supports students in doing higher-level tasks by ensuring that they understand the concepts they use. The system is limited to the comprehension level and is most effective for introductory courses in new subject domains such as programming, mathematics, or natural languages when students need to understand a significant number of new concepts and learn to handle them in typical situations. Worse-performing students gain more from such systems as the evaluation shows.

The main problem of the developed system is the small number of predefined questions compared to the number of subject-domain laws and possible errors, given that learning may require answering several questions concerning the same subject-domain law in a row. The properties of the developed formal models allow for large-scale question generation by mining existing open-source code which is the most important direction of further work. Another direction of further work is developing a method to build sets of follow-up questions from the formal description of the subject domain because creating and debugging them manually is a time-consuming process.

Author Contributions: Conceptualization, O.S.; Data curation, A.P.; Formal analysis, O.S. and A.A.; Funding acquisition, O.S. and A.A.; Investigation, M.D.; Methodology, O.S. and A.A.; Project administration, O.S.; Resources, A.A.; Software, N.P., M.D. and A.P.; Supervision, O.S.; Validation, A.A. and M.D.; Visualization, N.P. and M.D.; Writing—original draft, O.S., A.A. and M.D.; Writing—review & editing, O.S. All authors have read and agreed to the published version of the manuscript.

Funding: The reported study was funded by RFBR, project number 20-07-00764.

Institutional Review Board Statement: Ethical review and approval is not required, because this study involved the analysis of data being based on voluntary participation and having been properly anonymized. The research presents no risk of harm to subjects.

Informed Consent Statement: Informed consent was obtained from all students involved in the study.

Data Availability Statement: The source code of the developed system can be accessed at https://github.com/CompPrehension/CompPrehension (accessed on 5 November 2021).

Acknowledgments: The authors thank the students and teachers of Volgograd State Technical University who participated in the evaluation of the developed system, especially Elena Berisheva for organizing the process.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ASP	Answer Set Programming
AST	Abstract Syntax Tree
HTML	The HyperText Markup Language
ITS	Intelligent Tutoring System
IPTS	Intelligent Programming Tutoring System
LTI	Learning Tools Interoperability
SWRL	Semantic Web Rule Language

References

1. Bloom, B.S.; Engelhart, M.B.; Furst, E.J.; Hill, W.H.; Krathwohl, D.R. *Taxonomy of Educational Objectives. The Classification of Educational Goals. Handbook 1: Cognitive Domain*; Longmans Green: New York, NY, USA, 1956.
2. Girija, V. Pedagogical Transitions Using Blooms Taxonomy. *Int. J. Res. Eng. Soc. Sci.* **2019**, *9*, 492–499.
3. Ursani, A.A.; Memon, A.A.; Chowdhry, B.S. Bloom's Taxonomy as a Pedagogical Model for Signals and Systems. *Int. J. Electr. Eng. Educ.* **2014**, *51*, 162–173. [CrossRef]
4. Stanny, C. Reevaluating Bloom's Taxonomy: What Measurable Verbs Can and Cannot Say about Student Learning. *Educ. Sci.* **2016**, *6*, 37. [CrossRef]
5. Pikhart, M.; Klimova, B. Utilization of Linguistic Aspects of Bloom's Taxonomy in Blended Learning. *Educ. Sci.* **2019**, *9*, 235. [CrossRef]
6. Anderson, L.W.; Krathwohl, D.R.; Airasian, P.W.; Cruikshank, K.A.; Mayer, R.E.; Pintrich, P.R.; Raths, J.; Wittrock, M.C. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*, abridged ed.; Allyn & Bacon: Boston, MA, USA 2001.
7. Churches, A. Bloom's Digital Taxonomy. 2008. Available online: http://burtonslifelearning.pbworks.com/f/BloomDigitalTaxonomy2001.pdf (accessed on 5 November 2021).
8. Mitrovic, A.; Koedinger, K.R.; Martin, B. A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modeling. In *Proceedings of 9th International Conference on User Modeling, Johnstown, PA, USA, 22–26 June 2003*; Brusilovsky, P., Corbett, A., de Rosis, F., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; pp. 313–322. [CrossRef]
9. Taggart, M. Programming and Bloom's Taxonomy. 2017. Available online: https://theforeverstudent.com/cs-ed-week-part-3-programming-and-blooms-taxonomy-151cfc0d550f (accessed on 5 November 2021).
10. Omer, U.; Farooq, M.S.; Abid, A. Cognitive Learning Analytics Using Assessment Data and Concept Map: A Framework-Based Approach for Sustainability of Programming Courses. *Sustainability* **2020**, *12*, 6990. [CrossRef]
11. Yoo, J.; Pettey, C.; Seo, S.; Yoo, S. Teaching Programming Concepts Using Algorithm Tutor. In *EdMedia+ Innovate Learning*; Association for the Advancement of Computing in Education: Waynesville, NC, USA, 2010; pp. 3549–3559.
12. Skalka, J.; Drlík, M. Educational Model for Improving Programming Skills Based on Conceptual Microlearning Framework. In Proceedings of the International Conference on Interactive Collaborative Learning, Kos Island, Greece, 25–28 September 2018; Springer International Publishing: Cham, Switzerland, 2019; pp. 923–934. [CrossRef]
13. Skalka, J.; Drlik, M.; Benko, L.; Kapusta, J.; del Pino, J.C.R.; Smyrnova-Trybulska, E.; Stolinska, A.; Svec, P.; Turcinek, P. Conceptual Framework for Programming Skills Development Based on Microlearning and Automated Source Code Evaluation in Virtual Learning Environment. *Sustainability* **2021**, *13*, 3293. [CrossRef]
14. Sychev, O.; Denisov, M.; Terekhov, G. How It Works: Algorithms—A Tool for Developing an Understanding of Control Structures. In Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2, ITiCSE '21, Paderborn, Germany, 26 June–1 July 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 621–622. [CrossRef]
15. Sychev, O.A.; Anikin, A.; Denisov, M. Inference Engines Performance in Reasoning Tasks for Intelligent Tutoring Systems. In *Proceedings of the Computational Science and Its Applications—ICCSA 2021, Cagliari, Italy, 13–16 September 2021*; Gervasi, O., Murgante, B., Misra, S., Garau, C., Blečić, I., Taniar, D., Apduhan, B.O., Rocha, A.M.A., Tarantino, E., Torre, C.M., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 471–482. [CrossRef]
16. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
17. Ausin, M.; Azizsoltani, H.; Barnes, T.; Chi, M. Leveraging deep reinforcement learning for pedagogical policy induction in an intelligent tutoring system. In *Proceedings of the EDM 2019—Proceedings of the 12th International Conference on EducationalData Mining, Montreal, QC, Canada, 2–5 July 2019*; Lynch, C., Merceron, A., Desmarais, M., Nkambou, R., Eds.; International Educational Data Mining Society: Worcester, MA, USA, 2019; pp. 168–177.
18. Crow, T.; Luxton-Reilly, A.; Wuensche, B. Intelligent tutoring systems for programming education. In Proceedings of the 20th Australasian Computing Education Conference—ACE'18, Brisbane, QLD, Australia, 30 January–2 February 2018; ACM Press: New York, NY, USA, 2018. [CrossRef]
19. Kumar, A.N. Generation of problems, answers, grade, and feedback—Case study of a fully automated tutor. *J. Educ. Resour. Comput.* **2005**, *5*, 3. [CrossRef]

20. O'Rourke, E.; Butler, E.; Tolentino, A.D.; Popović, Z. Automatic Generation of Problems and Explanations for an Intelligent Algebra Tutor. In Proceedings of the 20th International Conference on Artificial Intelligence in Education, AIED, Chicago, IL, USA, 25–29 June 2019; Springer International Publishing: Cham, Switzerland, 2019; pp. 383–395. [CrossRef]
21. Brusilovsky, P.; Su, H.D. Adaptive Visualization Component of a Distributed Web-Based Adaptive Educational System. In *Intelligent Tutoring Systems*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 229–238. [CrossRef]
22. Fabic, G.V.F.; Mitrovic, A.; Neshatian, K. Adaptive Problem Selection in a Mobile Python Tutor. In *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization*; ACM: New York, NY, USA, 2018. [CrossRef]
23. Jeuring, J.; Gerdes, A.; Heeren, B. A Programming Tutor for Haskell. In Proceedings of the Selected Papers of the 4th Central European Functional Programming School, Budapest, Hungary, 14–24 June 2011; Springer: Berlin/Heidelberg, Germany, 2012; pp. 1–45. [CrossRef]
24. Lane, H.C.; VanLehn, K. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Comput. Sci. Educ.* **2005**, *15*, 183–201. [CrossRef]
25. Papadakis, S.; Kalogiannakis, M.; Zaranis, N. Developing fundamental programming concepts and computational thinking with ScratchJr in preschool education: A case study. *Int. J. Mob. Learn. Organ.* **2016**, *10*, 187. [CrossRef]
26. Price, T.W.; Dong, Y.; Lipovac, D. iSnap. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, 8–11 March 2017 ; ACM: New York, NY, USA,2017. [CrossRef]
27. Barra, E.; López-Pernas, S.; Alonso, Á.; Sánchez-Rada, J.F.; Gordillo, A.; Quemada, J. Automated Assessment in Programming Courses: A Case Study during the COVID-19 Era. *Sustainability* **2020**, *12*, 7451. [CrossRef]
28. Pillay, N. Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bull.* **2003**, *35*, 78–82. [CrossRef]
29. Polito, G.; Temperini, M. A gamified web based system for computer programming learning. *Comput. Educ. Artif. Intell.* **2021**, *2*, 100029. [CrossRef]
30. Sorva, J. Notional machines and introductory programming education. *ACM Trans. Comput. Educ.* **2013**, *13*, 1–31. [CrossRef]
31. Fincher, S.; Jeuring, J.; Miller, C.S.; Donaldson, P.; du Boulay, B.; Hauswirth, M.; Hellas, A.; Hermans, F.; Lewis, C.; Mühling, A.; Pearce, J.L.; Petersen, A. Notional Machines in Computing Education: The Education of Attention. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR '20, Trondheim, Norway, 15–19 June 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 21–50. [CrossRef]
32. Sorva, J.; Karavirta, V.; Malmi, L. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* **2013**, *13*, 1–64. [CrossRef]
33. Schoeman, M.; Gelderblom, H. The Effect of Students' Educational Background and Use of a Program Visualization Tool in Introductory Programming. In Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists on—SAICSIT'16, Johannesburg, South Africa, 26–28 September 2016; ACM Press: New York, NY, USA, 2016. [CrossRef]
34. Sychev, O.; Denisov, M.; Anikin, A. Verifying algorithm traces and fault reason determining using ontology reasoning. In *Proceedings of the ISWC 2020 Demos and Industry Tracks: From Novel Ideas to Industrial Practice Co-Located with 19th International Semantic Web Conference (ISWC 2020), Globally Online, 1–6 November 2020 (UTC)*; CEUR Workshop Proceedings; Taylor, K.L., Gonçalves, R., Lécué, F., Yan, J., Eds.; CEUR-WS.org: Aachen, Germany, 2020; Volume 2721, pp. 49–54.
35. Sychev, O.; Anikin, A.; Penskoy, N.; Denisov, M.; Prokudin, A. CompPrehension-Model-Based Intelligent Tutoring System on Comprehension Level. In Proceedings of the 17th International Conference on Intelligent Tutoring Systems, Virtual Event, 7–11 June 2021; Springer International Publishing: Cham, Switzerland, 2021; pp. 52–59. [CrossRef]
36. Kumar, A.N. Allowing Revisions While Providing Error-Flagging Support: Is More Better? In Proceedings of the 21st International Conference on Artificial Intelligence in Education, Ifrane, Morocco, 6–10 July 2020; Springer International Publishing: Cham, Switzerland, 2020; pp. 147–151. [CrossRef]
37. Kumar, A.N. Limiting the Number of Revisions while Providing Error-Flagging Support during Tests. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems, Chania, Crete, Greece, 14–18 June 2012*; Cerri, S.A., Clancey, W.J., Papadourakis, G., Panourgia, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 524–530. [CrossRef]
38. Kumar, A.N. Generation of Demand Feedback in Intelligent Tutors for Programming. In Proceedings of the 17th Conference of the Canadian Society for Computational Studies of Intelligence, Canadian AI 2004, London, ON, Canada, 17–19 May 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 444–448._35. [CrossRef]
39. Anikin, A.; Sychev, O. Ontology-Based Modelling for Learning on Bloom's Taxonomy Comprehension Level. In Proceedings of the Tenth Annual Meeting of the BICA Society, Seattle, WA, USA, 15–18 August 2019; Springer International Publishing: Cham, Switzerland 2019; pp. 22–27. [CrossRef]
40. Atzeni, M.; Atzori, M. CodeOntology: RDF-ization of Source Code. In *Proceedings of the 16th International Semantic Web Conference, Vienna, Austria, 21–25 October 2017*; d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., Heflin, J., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 20–28. [CrossRef]
41. Barana, A.; Marchisio, M.; Sacchet, M. Interactive Feedback for Learning Mathematics in a Digital Learning Environment. *Educ. Sci.* **2021**, *11*, 279. [CrossRef]
42. Elder, L.; Paul, R. The Role of Socratic Questioning in Thinking, Teaching, and Learning. *Clear. House J. Educ. Strateg. Issues Ideas* **1998**, *71*, 297–301. [CrossRef]

43. Yenmez, A.A.; Erbas, A.K.; Cakiroglu, E.; Cetinkaya, B.; Alacaci, C. Mathematics teachers' knowledge and skills about questioning in the context of modeling activities. *Teach. Dev.* **2018**, *22*, 497–518. [CrossRef]
44. Kumar, A.N. Solving Code-Tracing Problems and Its Effect on Code-Writing Skills Pertaining to Program Semantics. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education; Proceedings of the 20th Annual Conference on Innovation and Technology in Computer Science Education, Vilnius, Lithuania, 6–8 July 2015; Association for Computing Machinery: New York, NY, USA, 2015; pp. 314–319. [CrossRef]
45. Kumar, A.N. An Epistemic Model-Based Tutor for Imperative Programming. In *Proceedings of the 22nd International Conference on Artificial Intelligence in Education, Utrecht, The Netherlands, 14–18 June 2021*; Roll, I., McNamara, D., Sosnovsky, S., Luckin, R., Dimitrova, V., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 213–218. [CrossRef]
46. Schneider, L.N.; Meirovich, A. Student Guided Learning-from Teaching to E-learning. *Rev. Rom. Pentru Educ. Multidimens.* **2020**, *12*, 115–121. [CrossRef]
47. Burkšaitienė, N.; Lesčinskij, R.; Suchanova, J.; Šliogerienė, J. Self-Directedness for Sustainable Learning in University Studies: Lithuanian Students' Perspective. *Sustainability* **2021**, *13*, 9467. [CrossRef]
48. Chacon, I.A.; Barria-Pineda, J.; Akhuseyinoglu, K.; Sosnovsky, S.A.; Brusilovsky, P. Integrating Textbooks with Smart Interactive Content for Learning Programming. In *Proceedings of the Third International Workshop on Inteligent Textbooks 2021 Co-located with 22nd International Conference on Artificial Intelligence in Education (AIED 2021), Online, 15 June 2021*; CEUR Workshop Proceedings; Sosnovsky, S.A., Brusilovsky, P., Baraniuk, R.G., Lan, A.S., Eds.; CEUR-WS.org: Aachen, Germany 2021; Volume 2895, pp. 4–18.