ICCE 2006
14th International Conference on Computers in Education

Full paper

Problem Solving Process oriented Diagnosis in Logic Programming

Abstract. In this paper, we present the evaluation result of our constraint-based tutoring system for logic programming from which we derive the conclusion that students need diagnostic information and remedial hints corresponding to the stage of the problem solving process where they are stuck. For this reason, we propose a three steps diagnosis approach which consists of: diagnosis at the task analysis stage, diagnosis at the solution design stage and diagnosis at the implementation stage. Our diagnosis approach should not only help students learn logic programming, but also master the skills of task analysis and solution design.

Keywords: Constraint-based modelling, cognitive diagnosis, tutoring systems, logic programming, problem-solving process, instructional process.

Contact information
Nguyen-Thinh Le
University of Hamburg
Department of Informatics
Vogt-Kölln-Str. 30
22527 Hamburg
Tel. +49 (0)40 42883 2537
le@informatik.uni-hamburg.de

Problem Solving Process oriented Diagnosis in Logic Programming

Nguyen-Thinh Le, Wolfgang Menzel
Department of Informatics, University of Hamburg, Germany
le@informatik.uni-hamburg.de

Abstract. In this paper, we present the evaluation result of our constraint-based tutoring system for logic programming from which we derive the conclusion that students need diagnostic information and remedial hints corresponding to the stage of the problem solving process where they are stuck. For this reason, we propose a three steps diagnosis approach which consists of: diagnosis at the task analysis stage, diagnosis at the solution design stage and diagnosis at the implementation stage. Our diagnosis approach should not only help students learn logic programming, but also master the skills of task analysis and solution design.

Keywords: Constraint-based modeling, cognitive diagnosis, tutoring systems, logic programming, problem-solving process, instructional process.

Introduction

Error diagnosis plays an important role in an Intelligent Tutoring System (ITS) because diagnostic information is essential for modelling the state of student's knowledge and for initiating appropriate instructional actions. Currently, several programming tutoring systems apply a rather simple diagnostic approach by presenting a problem to the student and providing the possibility to submit a solution by choosing from several options or filling in a template. Diagnostic approaches supported by this type of solution submission might be used in tutoring systems aiming at helping students to become familiar with basic concepts of a programming language. In problem solving, however, students often make errors because they have difficulties with task analysis or solution design. Thus, at that point, diagnostic information about semantic or syntactic errors is not relevant. Rather it is important to consider the stage of the problem solving process, where the student becomes stuck: task analysis, solution design or implementation.

We have developed a web-based tutoring system for logic programming applying the constraint-based modeling (CBM) approach. In this paper, first, we introduce the constraint-based approach briefly. In the second section, we present the result of the preliminary evaluation of our system. From the analysis of the evaluation result, we claim that diagnostic information is only useful for students if it matches the stage of the problem solving process where the student has difficulties. In the third section we outline related work. Our diagnosis approach is introduced in the fourth section. The current state and further directions of our research are summarized in the last section.

1. Constraint-based Modeling

The CBM approach proposed in [1] can be applied to model general principles of a domain as a set of constraints. A constraint is represented as an ordered pair consisting of a

relevance part and a satisfaction part: `Constraint C = <relevance part, satisfaction part>`

The relevance part represents circumstances under which the constraint applies, and the satisfaction part represents a condition that needs to be fulfilled for the constraint to be satisfied. Constraints can be used to describe facts, principles or conditions which must hold for every solution contributed by the student. In addition, constraints can also be used to specify requirements of a task. Using the relevance part, constraints can be tailored according to an ideal solution, which represents the requirements of the given task. Requirements, which have to be satisfied in that specific situation, can be specified in the satisfaction part. More about the application of the CBM approach to model problem solving in logic programming can be found in [3, 4].

A constraint is evaluated by matching its relevance part to the solution. If the matching is successful then the solution should also fulfill the satisfaction part. Otherwise, the solution is considered to be incorrect with respect to the constraint that has been evaluated. If a constraint is violated, it indicates that the student solution does not obey principles of the domain or does not meet the requirements of the given task.

We developed a tutoring system (INCOM) [3] for logic programming applying the CBM technique. The diagnosis approach of our current system consists of two steps. It starts by hypothesising the Prolog pattern the student solution is based on. A Prolog pattern represents a solution strategy for a programming problem [4]. For a given programming problem, there are usually several appropriate patterns which can be applied to solve it. The pattern selection is carried out heuristically. The second step of the diagnosis examines whether the student solution satisfies the task requirements. In the evaluated version, we use constraints to model task requirements. If a constraint is violated, a programming technique has been applied incorrectly or a task requirement is not fulfilled.

2. Evaluation and the Problem of Remedial Hints

2.1 Evaluation results

We have conducted a preliminary evaluation for INCOM during the winter term 2004/05 at the University of Hamburg. We provided students with four exercise assignments:

1. Define a predicate which specifies the relationship between a list and its prefix.
2. Write a function to convert Peano numbers to integer numbers.
3. Write a predicate which defines an even Peano number.
4. Write a function to compute the sum of compound interest for a given amount, an interest rate and a duration in years.

Students have been requested to consult our system via a web interface when experiencing difficulties in solving those four exercise assignments. On our server machine, we registered 261 log files created by 99 users.

Table 1 number of false and correct trials for each task

Task	*trials/user*	*trials for a correct solution*	*task solved*	*task not solved*
1	6.07	4.33	11	7
2	6.21	6.54	22	23
3	5.72	6.83	27	17
4	6.21	74.5	1	24

The first goal of our evaluation was to find out which task is challenging for students. Table 1 provides the results of problem solving for each task. The 2nd and 3rd columns show, how many trials in average a user carried out, and how many trials he/she

needed to reach a correct solution. The last two columns tell us, how many log files contained correct solutions and how many did not. From the last two columns we can identify that most students could solve Task 1 and 3. For Task 2, the number of successful and unsuccessful attempts are almost the same. We also noticed that most users could not solve task 4. Analyzing the log files, we found two main reasons for these frequent failures:

1. Task 4 is more complicated than the other tasks. It includes the concept of recursion, arithmetic expressions, and arithmetic computation;
2. Many users were not able to derive a correct formula for the computation of a compound interest.

The second goal of our evaluation was to identify the problems of students and where they are usually stuck. By investigating the log files, we recognized that most errors have been detected by the first step of the diagnosis - the pattern identification process. Most users had one of the following problems:

- Users were not familiar with the data structure of Peano numbers. Some of them simply input "`Peano`" or `peano(X)` as arguments and expected that to be a Peano number.
- The arithmetic evaluation mechanism in Prolog poses considerable problems for many users. Some of them placed an arithmetic expression at an argument position and expected a functional evaluation. Others used "`=`" instead of "`is`" for arithmetic evaluation, as it is common in mathematical notations.
- Users called auxiliary predicates without defining them in the hope that they are built-in predicates. Or, they used arbitrary material at an argument or subgoal position and expected that the system is able to provide helpful hints.

Through errors detected during the second step of the diagnosis - the constraint evaluation process, we noticed that users had the following problems:

- Many users applied arithmetic expressions without making sure that the arguments are sufficiently instantiated. Sometimes, they transposed the positions of operands and result arguments or used operands not correctly.
- Instead of decomposing an input argument, many novice programmers composed it in recursive subgoals. Or, they decomposed an input argument and processed it, but then, they did not know how to return the result of the processed input value. This indicates that Prolog novices are not familiar with composition and decomposition.

The third goal of our evaluation was to determine the efficacy of the system. Our system provides three levels of feedback. If the system detects an error in the user's solution, first, it notifies that the solution is incorrect; second, upon request, it shows the problem location, gives an explanation and third, it provides suggestions to remove the error. We evaluate the efficacy of the system by determining if errors disappeared after users have seen the error location or a remedial hint. A remedial hint includes an error explanation and a correction proposal.

In 75,6 % of a total of 632 false trials, students were interested in system feedback. In 60,5% of 478 feedback requests, after seeing the error location, they requested more detailed error explanation and remedial hints. That indicates that most students are interested in receiving feedback from the system in order to improve their solutions. In 68.3% of cases after seeing the error location without requesting remedial hint, users were able to remove the error. In 75.8% of cases after requesting remedial hints, the error was eliminated. As expected, the efficacy of remedial hints is higher than that of error location because remedial hints give more information. The result shows that in general the system is helpful for students. However, the efficacy of our current system does not satisfy our ambitions.

We investigated the log files to trace back how students have corrected their solutions after having read the feedback. We noticed that students could not correct their solutions according to some remedial hints provided by our system. This can be attributed 1) to the incoherence of the remedial hints which are specified for isolated constraints [5, 6] and 2) to a

mismatch between the feedback and the stage of the problem solving process where the student's difficulty occurred. In this paper, we mainly address the second problem.

2.2 The problem of solution remediation

We illustrate the problem of providing students with appropriate feedback with the following example. The third exercise assignment requests students to define a predicate which specifies the relationship between a list and its prefix. Our system expects a correct solution like IP1 or IP2 which uses an auxiliary predicate `append`.

Task: define a predicate to examine the relationship between a list and its prefix
Solution IP1:

```
prefix([], _).
prefix([H|R], [H|T]):-prefix(R,T).
```

Solution IP2:
IP2: `prefix(L1, L2):-append(L1, Rest, L2).`

A student submitted the following solution SP1 for the task above. Our system hypothesizes that the student decided to apply the strategy IP1, it then evaluates the relevant constraints and returns the corresponding diagnostic information.

Student solution SP1:

```
prefix(List, List).
prefix([], List).
```

Remedial hints:
Error1: a base case in your solution is superfluous.
Error2: a recursive case in your solution is missing.

The solution indicates that the student is in a position to specify a base case *"prefix([], List)."* but not able to specify a recursive case. Perhaps the student wanted to specify the type restriction for the argument positions by giving the clause "*prefix(List, List)*". More likely, however, it is that the student does not know how to specify a list data structure which is required for both predicate arguments. That means he/she is not able to fully analyze the task and to specify the arguments correctly. Therefore, remedial hints concerning solution design are not helpful for the author of the solution above. In this case, we need to help the student analyze the task requirements. The task analysis includes questions like: Which information should be represented as an argument? What kind of data structures should be specified for predicate arguments? Which mode should an argument have?

The following student's solution SP2 indicates that he/she has succeeded with the task analysis, but is now struggling with designing a solution for the given task.

Student solution SP2:

```
prefix([X], [X]).
prefix(L, [X|Rest]):- append(H, Rest, [X|Rest]),prefix(H, Rest).
```

Remedial hints:
Error1: the subgoal prefix(H, Rest) is superfluous.
Error2: the clause prefix([X], [X]) is superfluous.
Error3: L should be unified with H.
Error4: the argument [X|Rest] in the head of 2nd clause should be represented as a variable.

The system hypothesizes that the student was following strategy IP2 and evaluates the corresponding constraints. While the first two feedback messages concern the solution design corresponding to the strategy IP2, the last two consider the erroneous implementation of arguments. This might have caused the student to be confused because she/he is currently having problems with designing the solution, not with the

implementation. At this stage, the student has to deal with the questions: what kind of clauses and subgoals are required to construct a solution according to the intended design strategy, and how these clauses and subgoals have to be arranged? Table 2 tells us the proportion of false attempts due to errors in task analysis and in solution design. The second and the third columns indicate the absolute number of students' attempts to solve the tasks. We notice that students made most errors (70%) at the stage of analyzing Task 3. That means, they were not able to specify a Peano number correctly. We also see that students had difficulties with designing solutions for Task 4. 42% of their attempts for Task 4 were not successful at finding appropriate clauses or subgoals.

Table 2 Proportion of false attempts due to errors in task analysis and in solution design.

Task	*Total attempts*	*False attempts*	*Errors in task analysis*	*Errors in solution design*
1	91	70	7%	20%
2	242	205	25%	18%
3	246	210	70%	17%
4	149	147	9%	42%

From the evaluation result and the analysis of student's programs above, we can derive the need for a diagnosis approach which is able to provide diagnostic information corresponding to the stage of the problem solving process.

3. Related works

Available tutoring systems are able to detect semantic or syntactic errors in a program. However, this kind of diagnostic information is not useful for students who already have difficulties in the early phases of problem solving. The problem is to determine which level of understanding the student has and how to guide him/her to correct his/her solution in a way he/she is supposed to do. Various attempts have been developed in this direction, but none of them is really able to provide diagnostic information tailored to the stage where the difficulties occurred. The Pascal tutoring system [7] is able to infer the student's intention and to diagnose errors by mapping a student program to programming plans. This system focuses the diagnosis mainly on the solution design applying programming plans and misses the diagnosis at the task analysis stage. A model-tracing tutor [8, 9] follows the student's intention by forcing the student to act as an expert would do. Hence, a model-tracing tutor always pretends to know the student's intention. However, model tracing does not guarantee that student errors can always be corrected. When a student performs an act, which is neither on a correct path nor on a anticipated incorrect one, model tracing has nothing to say other than that is probably incorrect [10]. ELM-PE provides a syntax-based structure editor, which guides the student filling in appropriate insertions into predefined LISP statement slots, such that only valid LISP expressions may be constructed [11]. The diagnostic approaches mentioned restrict students' creativity and do not support them to improve the problem solving skill.

Some other approaches introduce different abstraction levels of errors made by students. The approach in [12] represents student's actions and errors in terms of knowledge applied in a learning context. Two levels of knowledge are differentiated. The micro-level contains elements describing problems, operators, and control structures and the macro-level describes conceptions. The micro-level represents the way a conception may be revealed by a student, whereas the macro-level represents conceptions in terms of knowledge. The diagnosis approach is driven by taking into account student's actions related to a particular task and the system provides explanations on the student's reasoning by recognizing sub-jacent knowledge. According to [12], an environment that intends to

provide personalized feedback must be able to interpret student's actions in terms of knowledge. The approach in [13] distinguishes the surface level student model from a deeper level student model. The former one represents the scheduled problem solving plans and applied procedural knowledge. The authors of [13] argue that just diagnosing problem solving knowledge applied by the student is not sufficient, because the sequence of the procedures the student has used may reflect his or her belief in the domain axioms. Therefore, it is necessary to build a deeper level student model which consists of diagnostic hypotheses explaining the procedural operations of a student in terms of the domain axioms. Both approaches [12, 13] introduce different levels of knowledge which can be inferred from the student's input. However, they do not provide diagnostic information along the process of problem solving.

We propose a diagnostic approach which not only enables students to input a solution for a given task in free form. It also supports the students at all three stages of the problem solving process[1]: task analysis, solution design and implementation.

4. Three Steps Diagnosis in Logic Programming

4.1 Diagnosis at the task analysis stage

To create a logic program, first, it is necessary to know how many arguments are required to solve the given task. Normally the number of required arguments can be inferred from the task specification. The student should be able to understand the functionality of every argument which is used to define a predicate. If an argument does not have any function, it is considered to be superfluous. If information from the given task has not been modeled as an argument in the predicate definition, then the student has missed a necessary argument to solve the given task. The second step of the task analysis is to determine the argument modes. In logic programming, an argument can have input mode, output mode or both. Students are requested to specify a mode for each argument of the predicate to be defined according to the given task. The last step of task analysis is the definition of appropriate data structures for the argument positions. A data structure for an argument in logic programming can be an atom, a number, a list or a special kind of term (i.e. Peano number).

We request students to input the information for predicate declaration, i.e. argument list, argument mode and data structure, before they submit a complete solution. The system examines their inputs of the task analysis as the following example shows:

Task description: *please define a predicate which specifies the relationship between a list and its prefix. You can use the built-in predicate* `append` *if necessary.*
System: *please, input the predicate name and the list of arguments with the appropriate modes.*
Student: `prefix(?List1, -List2)`
System: *Error location: 2nd argument;*
Explanation: wrong mode;
Suggestion: check the task specification and choose the appropriate mode.
Student: `prefix(?List1, ?List2)`
System: *The declaration for the predicate is correct.*

4.2 Diagnosis at the solution design stage

Now, the student has analyzed the task correctly. The system allows the student to put in a solution. A template `prefix(List1, List2)` according to the predicate declaration is

[1] Our term "problem solving process" differs from the definition in [13] where a sequence of operations applied by the student in solving a given problem is called a problem solving process.

given to the student. After the student has submitted her/his solution, the system hypothesizes the design strategy of the student [4] and returns appropriate feedback.

System: *please, input your solution for the task above.*
Student: `prefix([X],[X]).`
`prefix(L,[X|Rest]):-append(H,Rest,[X|Rest]),prefix(H,Rest).`
System: *we hypothesize that you apply the built-in predicate* `append` *to concatenate the first list with an arbitrary second list to get the third list. Please, consider the following feedback:*
Error1: Location: clause `prefix([X],[X])`
Explanation: this clause is superfluous. It is not necessary for this design strategy.
Remedial hint: please remove it.
Error2: Location: `prefix(H,Rest)`
Explanation: this subgoal is superfluous. It is not required for this design strategy.
Suggestion: please remove it.
Student: `prefix(L,[X|Rest]):-append(H,Rest,[X|Rest]).`

4.3 Diagnosis at the solution implementation stage

Once, the student's solution conforms with the design strategy the system hypothesized, the system moves the diagnosis to the implementation level. This step deals with the questions: How is a necessary subgoal implemented? How are the arguments in the student's solution implemented? Our system replies to the last student's input in the example above as follows:
System: *please, consider the following feedback*
Error 1:Location: L, H
Explanation: We need to process L. Therefore L and H have to be co-referenced.
Remedial hint: write a subgoal L=H, or replace H by L, or replace L by H.
Error 2: Location: [X|Rest] in the head of clause 1
Explanation: This argument position needs a variable which represents a list.
Suggestion: choose a variable name.
Error 3:Location: [X|Rest] in subgoal 1, clause 1
Explanation: This argument needs to represent the concatenation of two lists.
Suggestion: choose a variable name instead using a list representation

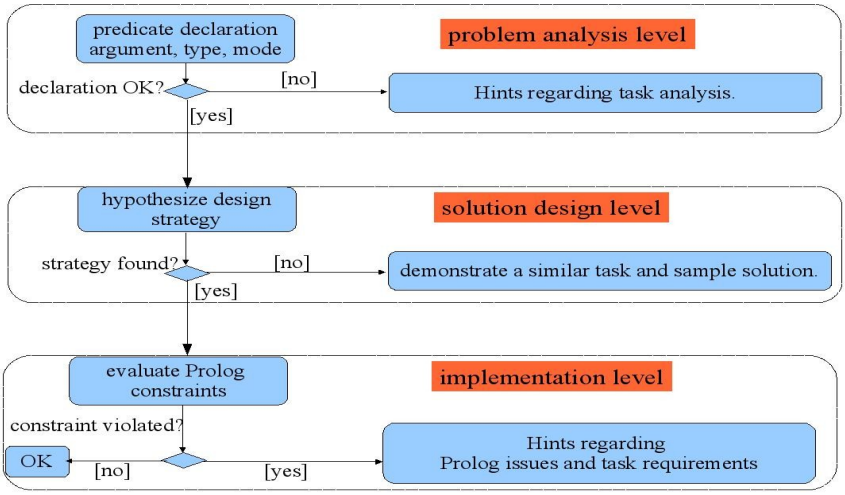

Figure 1: The three steps of the diagnosis process

A subgoal can be: unification, de/composition, calculation, an arithmetic test, binding, relation, recursion or user defined. An argument can be a variable, anonymous variable, list, atom (letter), number (float, int), Peano number, arithmetic expression or arbitrary term. For each subgoal, there are different implementation techniques. For

example, to implement an arithmetic test subgoal “less then”, we have three possibilities: Y<Y, \+X>=Y or Y>X. The whole diagnosis process is illustrated by Figure 1.

5. Conclusion and Future Work

We have presented the evaluation of our current tutoring system for logic programming. The diagnosis component of this system is developed applying the constraint-based modeling approach. The evaluation result indicated that students have not only difficulties on the on the implementation level, but also on the task analysis level (e.g. data structure for Peano numbers) and the solution design level (e.g. de/composition of lists). The evaluation result has shown that it is necessary to devise a diagnosis approach which is able to deliver diagnostic information corresponding to the stage of the problem solving process where the student is stuck. For this purpose, we have proposed a three step diagnosis approach: 1) diagnosis at the problem analysis stage, 2) diagnosis at the solution design stage and 3) diagnosis at the implementation stage.

The three step diagnosis approach can serve several educational purposes. First, it helps students master analysis skills. Second, it supports students in solving programming problems by using design strategies and lastly, students become familiar with the semantics of logic programming. The diagnostic component of our system is under restructuring. We plan to launch and to evaluate a second version of our tutoring system during the winter term 2006/2007.

References

[1] Ohlsson, S. (1994). *Constraint-based Student Modeling.* In J. E. Greer, G.I. McCalla, Student Modelling: The Key to Indivi-dualized Knowledge-based Instruction, 167-189. Berlin.

[2] Suraweera, P., Mitrovic, A., and Martin, B. (2005) *A Knowledge Acquisition System for Constraint-based Intelligent Tutoring Systems.* http://www.cosc.canterbury.ac.nz/tanja.mitrovic/Suraweera-AIED05.pdf

[3] Le, N.T., and Menzel, W. (2005) *Constraint-based Error Diagnosis in Logic Programming*. In Proceedings of the 13th International Conference on Computers in Education.

[4] Le, N.T. (2006) *Using Prolog Design Patterns to Support Constraint-Based Error Diagnosis in Logic Programming.* ITS workshop on "Intelligent Tutoring Systems for Ill-Defined Domains" (accepted to be published).

[5] Kodaganallur, V., Weitz, R.R., and Rosenthal, D. (2005) *A Comparison of Model-Tracing and Constraint-based Intelligent Tutoring Paradigms.* In International Journal of Artificial Intelligence in Education, Vol. 15, 117-144.

[6] Menzel, W. (2006) *Constraint-based Modeling and Ambiguity*. In International Journal of Artificial Intelligence in Education, Vol. 16, Nr. 1.

[7] Johnson, W. (1986) *Intention-based Diagnosis of Novice Programming Errors*. Morgan Kaufmann.

[8] Anderson, J.R., and Reiser, B. (1985) *The LISP Tutor.* Byte, 10, 159-175.

[9] Anderson, J.R., Conrad, F.G., and Corbett, A.T. (1989) *Skill Acquisition and the LISP Tutor.* Cognitive Science 13, 467-505.

[10] Martin, B. (2001) *Intelligent Tutoring Systems: The Practical Implementation of Constraint-based Modelling.* PhD thesis, University of Canterbury.

[11] Weber, G., and Möllenberg, A. (1995) *ELM Programming Environment: a Tutoring System for LISP Beginners*. In Cognition and Computer programming, 373-408.

[12] Webber, C. (2004) *From Errors to Conceptions – an Approach to Student Diagnosis.* In Proceedings of the 7th International Conference on Intelligent Tutoring Systems.

[13] Matsuda, N. and Okamoto, T. (1992) Student Model Diagnosis for Adaptive Instruction in ITS. In Proceedings of the 2nd International Conference on Intelligent Tutoring Systems., 467-474.