Constraint-based Error Diagnosis in Logic Programming

Nguyen-Thinh Le[a], Wolfgang Menzel[b]
[a,b]*Department of Informatics, University of Hamburg, Germany*
{le, menzel}@informatik.uni-hamburg.de

Abstract. Using the constraint-based modeling approach, we have developed a diagnostic component, which is able to identify errors made by learners of a logic programming language when implementing a given task specification. It uses patterns to hypothesize the intention of a learner and programming techniques to model conditions on the semantic well-formedness of the program code. These conditions are expressed by means of constraints, which are evaluated on the student solution. Guiding feedback can be derived from constraint violations and is presented to the student with different degrees of informativity. The component has been integrated into a web-based tutoring system and tested on a number of exercises by the participants of an introductory course in logic programming.

Keywords: Constraint-based error diagnosis, Tutoring systems, Logic programming.

Introduction

Error diagnosis is one of the essential components of Intelligent Tutoring Systems because understanding the current difficulties of a student is indispensable for providing him with guiding help. This information is necessary irrespective of how this feedback is going to be presented. Thus, error diagnosis and feedback generation can and should be separated to a certain degree. Numerous approaches to error diagnosis in programming languages have been proposed. Many of them turned out difficult to apply, do not provide enough diagnostic information or are restricted to a particular pedagogic strategy. In this paper, we present our experience in developing a constraint-based error diagnosis component for logic programming and highlight its strengths and weaknesses.

The constraint-based approach was introduced by Ohlsson in [1] and has been proven successfully in building Intelligent Tutoring Systems (ITS) in the domains of SQL and database design [2]. Since a logic programming language can be understood as the relational calculus enriched with recursion and function symbols, the question arises naturally, whether constraint-based diagnosis techniques can also be used in the more general case of a declarative programming language. To investigate this issue, we developed a diagnosis component and integrated it in a web-based tutoring system (INCOM). This system is intended to help first year university students to overcome difficulties while doing their homework assignments in programming with Prolog. Students are offered a database of Prolog exercises. They can select from this database the exercises, for which they are in need of help. The system attempts to diagnose their solution and returns feedback which indicates possibilities for remedy. Thus, the students can improve their solutions successively.

Three main issues have been identified as being crucial for designing an error diagnosis system. First, we have to understand the learning domain, namely logic programming and its subjects i.e. database query and recursion. Second, we need a diagnosis method which can deliver three kinds of information about an error: where the error is, what kind of error it is and how it could be remedied. The third issue is the role didactics plays in the design of a tutoring system. Here we can derive some inspiration from how a human tutor proceeds when correcting a program solution.

First the human tutor observes the structure of a student's solution and tries to guess, what kind of approach the student is following. We refer to a particular way to solve a programming problem as a programming pattern. Patterns can be derived from experience or a systematic study of the program structure. They impose several conditions that must be fulfilled in order to ensure the semantic correctness of the program code. By examining these conditions, the tutor can look deeper into the student's solution and seek for possible misconceptions. If a condition is not fulfilled, the tutor marks the erroneous position and writes his feedback beside the error.

Based on this scenario, we have developed our constraint-based approach. Firstly, it tries to "guess" the pattern the student is following, based on a generalized description of the corresponding program structure. Thereafter, it examines the semantic correctness of a solution by evaluating programming techniques applied to this pattern. The correctness of programming techniques is described by constraints. If a constraint is violated, a technique is not correctly used and an appropriate remedial hint is passed back to the user.

In the next section, we briefly overview the state of the art. In the second section, we describe how we combine patterns, Prolog programming techniques and constraint evaluation in the design of a coherent diagnosis component. The fourth section introduce the architecture of INCOM. Finally, the paper illustrates future directions of our research.

1. State of Art

Over the last two decades, numerous error diagnosis approaches in the domain of programming languages have been devised, such as program transformation [3,4,5], program verification [6], plan and bug library [7,8,9,10], model tracing [11] and constraint-based modeling [12]. Among these, model tracing is used by cognitive tutors which are some of the most successful ITS today [13].

The model tracing approach keeps track of the student's programming process and tries to guide him towards expert programming behavior. Possible actions a student might take are described by means of production rules. The set of production rules includes buggy rules, which represent erroneous actions that would be taken by a novice and ideal rules, which represent expert programming skills. By tracing the actions of the student with a collection of these rules, model tracing systems are able to build a student model and intervene in the programming process of the student. Whenever he performs a “bad” action, the system can return feedback immediately. Anderson and Reiser [11] applied this approach to build a tutor system for Lisp.

While the model tracing technique has been widely applied for developing cognitive tutor systems [14], recently, the constraint-based technique has been showing great promise as an alternative, which focuses on static cognitive states rather than problem solving processes [15]. This technique has been employed successfully to build an SQL tutor system [2] and has also been researched in the domain of data structures [16]. A number of factors explain why the constraint-based approach seems to be auspicious:

- It is relatively easy to model domain knowledge by means of constraints. It does not require an inference mechanism [15].
- A separate expert model is not necessary because the expert information is encapsulated in the constraints [1].
- This approach is more tolerant than model tracing with regard to the incompleteness of the knowledge base. It can recognize a correct solution submitted by the student, even if

that solution is different from the ideal one. If no constraint is violated, then the student's solution is still considered to be correct with respect to the notion of correctness embodied in the constraint base [12].
- It is neutral with respect to the tutor strategy. It provides a description of the error in terms of the constraints which the student has violated, but it leaves open the question of what instruction is implied by that description. The error description can be available immediately or later in a summarized form [1].

2. The pattern and constraint-based error diagnosis approach

2.1 Patterns

A pattern is a standard way to solve a recurrent problem. For example, many people multiply two numbers using the method of decimal-offsets. There are, of course, other ways to carry out such a computation, e.g. the Russian peasant algorithm. It reduces multiplication to four elementary operations: doubling a number, dividing a number by two, subtracting one, and adding two numbers.

Even after the basic algorithmic idea has been chosen, there are still different ways to implement it, e.g. a recursive or an iterative solution. Analog to arithmetics, such typical problems and standard ways to solve them can also be identified in other areas of programming. We call them patterns.

In Prolog, a pattern includes a general structure and a number of programming techniques applied to this structure. For example, the two predicate definitions below, `member/2` and `nested_list/1`, follow the so-called pattern "test-for-existence" [17].

```
member/2                            nested_list/1
member(H,[H|T]).                    nested_list([H|T]):-islist(H).
member(P,[H|T]):-member(P,T).       nested_list([H|T]):-nested_list(T).
```

The pattern "test for existence" determines that some collection of objects has at least one object with a specified property, i.e a list of terms has at least one term which is also a list. We generalize the structures of the two predicate definitions above and specify a structure for this pattern as follows:

```
pred(<<V1>>, [V2|V3]):-subgoal(1).
pred(<<V4>>, [V5|V6]):-pred(<<V7>>, V8).
```

In this representation, the expression `<<X>>` stands for a specific number of arguments and `subgoal(Y)` is replaced by a task dependent subgoal. This generalized structure for "test-for-existence" is accompanied by two programming techniques which establish the semantics of the pattern and are refered to as pattern specific programming techniques. First, as the first n arguments are used to determine that a single input element has the desired property, they represent the information which will not be changed during the computation process. These arguments must be used according to the so-called "***same***" technique. It requires the arguments in `<<V4>>` and `<<V7>>` to be co-referenced, i.e. to share the same value. The second technique applied to the last argument of the second clause is the "***list head***" technique which requires that `V6` and `V8` must have the same value [18]. In addition to the four patterns described by Brna which are mainly for list processing tasks [17], we defined some patterns to handle structures like Peano numbers.

In order to create a reference definition for `member/2` which follows the pattern "test-for-existence", an instance of the appropriate pattern is generated. Then, `<<V1>>`, `<<V4>>`, `<<V7>>` are replaced by variables `V1`, `V4`, `V7` and `subgoal(1)` is replaced by an empty string which means that the body of the first clause contains no subgoals. Thus, we obtain the following reference structure for `member/1`:

```
    pred(V1, [V2|V3]).
    pred(V4, [V5|V6]):-pred(V7, V8).
```
where in addition to pattern specific techniques, a task specific technique is applied to the first clause which represents a base case and requires that the property argument is a member of a list. That means `V1` is equal to `V2`.

Gegg-Harrison [19] defined a set of fourteen Prolog standard structures which are called schemata. A schema is specified based on a class of programs which share a common underlying structure and exemplify general techniques. He argued that the syntactic structure and the semantic interpretation of two Prolog programs are highly related and this makes it possible to compare the semantic similarity between two Prolog programs by comparing the similarities of their syntactic structures. This method, however, fails in cases, where the subgoals can be transposed without affecting the semantics. Hong [20] accepted that basic Prolog schemata are useful for presenting the general idea of techniques to the student. But, a basic Prolog schema is less useful for recognizing a student program since it does not provide much grammatical information.

Contrary to schemata, a pattern is described by a general structure (a schema) and a set of programming techniques which ensure the semantic correctness of a class of programs. Schemata have been proposed to teach Prolog by providing the novice Prolog programer with a template with place-holders for completion [19,21] and for program transformation [20]. We use patterns to diagnose errors in a student's solution.

2.2 Constraints

Techniques capture semantic relationships between variables within a clause. As such, they say something about the computation being undertaken rather than simply providing a syntactic pattern. A technique is language dependent (i.e., Prolog), but task independent, e.g. the same technique might be used in sorting a list or in finding the maximum of two numbers. Furthermore, a technique might apply to only part of a complete procedure, and many techniques may be combined together in a procedure [18,22,23]. Brna [22] raised some questions about the representation of techniques. How might techniques be specified? Which structural elements we really care about?

We address these problems by applying the constraint-based modeling approach. A constraint consists of two parts: a relevance and a satisfaction part [1]. The first part identifies the structural elements, for which a constraint is relevant. The latter examines if these elements satisfy the conditions of a constraint. For instance, the following statement can be described by a constraint:

"*if the solution follows the pattern 'test-for-existence', then a '**same**' technique must be applied to the arguments which represent a property and a '**list head**' technique must be applied to the input list.*"

The "if" phrase corresponds to the relevance part and the "then" phrase to the satisfaction part of the constraint. The statement mentioned can be separated into two constraints which have to be evaluated in conjunction. Hence, constraints express units of domain knowledge and can be used to describe the semantic requirements of techniques. In our system, we distinguish pattern specific from task specific constraints.

The relevance part of a constraint is the pattern a solution applies and the satisfaction part is a semantic condition for a programming technique or a combination of them. The following types of conditions for the satisfaction part are required for our diagnosis component and can be used under the assumption that a pattern has been identified:

- `arg_value(V1,V2)`: value of `V1` and `V2` must be the same.
- `arg_value(V1,<Constant>)`: `V1` has a constant value.
- `type_value(V1,<type>)`: argument `V1` must have type `<type>` which can be number, atom and list.

- `op_value(Op,<operator>)`: operator `Op` must be equal to `<operator>`.
- `before(<item1>,<item2>)`: `<item1>` must appear before `<item2>`. An item can be an argument, a subgoal or a clause.
- `and_value, or_value, not_value`: conjunction, disjunction, negation of constraints.

In the INCOM system, constraints are described by XML expressions. The piece of XML code below describes a satisfaction part of a constraint which is valid for the pattern "test-for-existence". It specifies that the variables `V1` and `V2` must be the same. If not, an error of type "same_argument" will be raised and the corresponding feedback is made available. The cost for this violated constraint is 3. This cost is used by the Feedback Generator component to sort error messages according to the severity of errors. Moreover, it is exploited by the pattern identification process to find the pattern with the least cost. Table 1 defines five degrees of severity for constraints.

```
<constraint>
   <constraintcontent>
         <constraintGround>
         arg_value('V1','V2')
         </constraintGround>
   </constraintcontent>
   <constrainterror>
       <constrainterrortype>
         same_argument
       </constrainterrortype>
       <constrainterrortext>
         'You should apply the "same" technique.'
       </constrainterrortext>
   </constrainterror>
   <constraintpenalty>3</constraintpenalty>
</constraint>
```

Table 1: Costs for constraints

Cost	Constraint
1	which just gives information.
2	which makes some suggestion, e.g. to improve performance.
3	error on the argument and operator level
4	error on the subgoal and clause level. E.g. the order of two subgoals is wrong
5	indicates that information of the exercise description has been overlooked by users. Missing information is directly highlighted in the exercise description.

2.3 A two-step diagnosis: Pattern identification and Constraint evaluation

As there are usually some patterns for solving a programming problem, the first step of our diagnosis needs to identify the pattern underlying the student's solution. We use it as an hypothesis about the intention of the student.

Beginning with the first pattern, a reference solution structure is instantiated based on the selected pattern. The student's solution is transformed to an internal format. Then the two structures need to be matched. This process carries out a heuristic search to map clause to clause, head to head, subgoal to subgoal, argument to argument and operator to operator of the two structures. Unmatched elements of a structure incur a cost which is highest for unmatched clauses. The cost for unmatched structures on the subgoal and head level is higher than on the argument and operator level. This means that the errors which occur on the argument and operator level are more easily tolerated than on the clause level. The matching process results in a clause map, a subgoal map, and a variable map which contain pairs of clauses, subgoals, arguments and operators from the student's solution and the reference structure, respectively. Also, a type map is created which contains type information of arguments and predicates in the student's solution. Currently, our system is able to determine if an argument is a list, an atom, a number or can have any type. As we decompose a predicate solution into clauses, head, subgoals, arguments and operators for matching purposes, we are able to assign positions of these structural elements with position numbers which are also included in the resulted maps. The reference structure of a pattern, which can be matched against the student solution with the least cost, is taken to be the most plausible hypothesis for the pattern the student followed. After the matching process is finished, the constraint analyser is going to evaluate the pattern specific and the task specific constraints.

Information from the clause map, subgoal map, variable map and type map can be used in order to evaluate the satisfaction part of constraints. For example, we can take advantage of the variable map to evaluate `arg_value(V1,<Constant>)` to determine if a pattern variable is bound to a program constant. The type map provides information for evaluating `type_value(V1,<type>)`. The position information in a variable or clause map allows examining the succession of two items.

If a constraint is violated, the constraint analyser forwards position information about the affected elements and the corresponding error text to the Feedback Generator. How these components of our diagnosis system interact, it is described in the next section.

3. Architecture

INCOM is a web-based system. Its architecture is divided into three layers (figure 1). The first layer contains the student interface which provides the exercise description and offers a means for students to input their solution. Users can choose one of the available tasks and input their entire solution. In addition, the student interface layer can present information about errors in the student's solution. The second layer comprises the diagnosis component and the feedback generator. The diagnosis component is separated into two sub-components. The first one performs the pattern identification and the second one evaluates the constraints. Errors which are found during the pattern identification and constraint analyzing process will be passed to the feedback generator component. The third layer inhabits the knowledge base of the system. In this component, the exercises and their solution space are described.

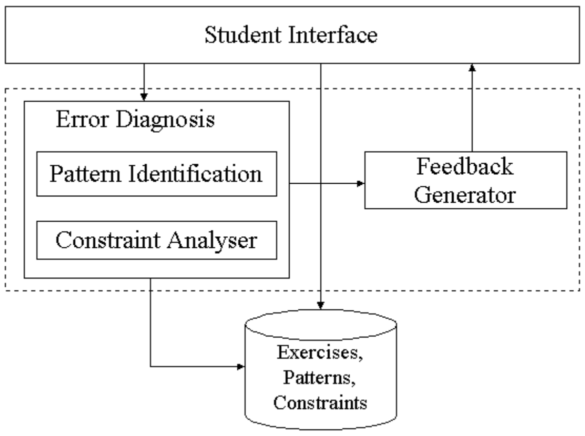

Figure 1: Architecture of INCOM

The knowledge base is stored as an XML file, which is separated into two parts. The first one is a collection of possible patterns. In the second part, the exercises are specified. Each exercise has its description and a list of applicable patterns. For each applicable pattern, the reference structure is specialized with additional exercise specific rules.

The diagnosis component begins with detecting syntax errors. For that purpose, the student's solution is forwarded to the Prolog compiler. In principle, conditions on the syntactic well-formedness of a predicate definition could also be modeled by means of constraints, but the focus of our research has been on the semantic and pragmatic aspects. If the syntax of the student's solution is correct, the program is normalized into an internal structure and the diagnosis is invoked. Otherwise, the diagnostic information of the compiler is returned to the student interface. The pattern identification component finds the best pattern and the constraint analyzer evaluates the pattern specific and the exercise specific constraints as described in section 2.

The errors collected during the diagnosis process are passed to the Feedback Generator which relies on an established collection of error types. For each error type, an error explanation is specified. The error types, which are produced by the structure matching process, are accompanied with a general remedial hint, which is exercise independent. Errors of this type are e.g. additional, missing or alternated arguments, subgoals and clauses. For errors which are detected by a constraint violation, the remedial hint needs to be specified by a human tutor individually when he authors the exercises. Thus, the Feedback Generator can provide position, explanation and remedial hint for errors if requested by the student.

4. Conclusion and Future works

When viewed as an intention-based program diagnoser, INCOM would be related to the PROUST [10], APROPOS2 [7] and MEDD [24] which are also intention-based. The most important difference between INCOM and other intentional diagnosers is that the others use bug library, whereas INCOM gives the exercise author a possibility to constrain the semantics of a solution object.

A preliminary evaluation has been carried out during the winter term 2004/05 at the University of Hamburg. System use was not mandatory, but recommended in case the student wanted additional help. Deliberately, we did not use an authentication mechanism. Hence, students could log in under different names. No reliable user identification was possible and even not desired to respect the privacy of students and encourage group work.

Currently, we have 261 log data created by 99 distinct users. The result is that in 68.3% of the cases after requesting error location without remedial hints and 75.8% of the cases after requesting remedial hints users were able to remove the indicated error [25].

Our experience has shown that the constraint-based approach can also be applied to model domain knowledge in a logic programming language. With a small set of constraint types, we were able to model the solution space for a number of typical recursion exercises in Prolog. The main difficulty we have been faced with was the definition of the patterns. If a pattern is defined for too broad for a class of programs, it is very time consuming to find appropriate specialization rules for every corresponding exercise. If a pattern is too specific, we have to define too many of them. Here, a proper balance has to be found. In the future, we also want to extend the system by defining patterns for predicates which use arithmetics. In addition, we are also planning to develop a tutor interface for authoring exercises. Besides the existing patterns which can be used to create new exercises, additional patterns can then be developed using this interface.

Acknowledgement

This research was funded by the E-Learning Consortium Hamburg, grant number A0035.

References

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

[2] A. Mitrovic et al. Constraint-based tutors: a success story. In L. Monostori and J. Vancza, *Proc. of the 14th Int. Conf. on Industrial Eng. App. of AI and Expert Systems,* 931-940, Budapest, 2001.

[3] A. Adam and J. Laurent. Laura, a system to debug student programs. *AI*, 15, 75-122, 1980.

[4] P. Vanneste. *A Reverse Engineering Approach to Novice Program Analysis*. PhD thesis, KU Leuven Campus Kortrijk, 1994.

[5] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering,* 29(4):360-384, 2003.

[6] W. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Los Altos, CA: Morgan Kaufmann, 1988.

[7] C.K. Looi. Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System. *Instructional Science*, 20, 215-263, 1991.

[8] L. Wills. Automated program recognition: A feasibility demonstration. *AI,* (45), 113-171, 1990.

[9] G. Weber. Episodic learner modeling. *Cognitive Science,* (20), 195-236, 1996.

[10] W. Johnson. *Intension-based Diagnosis of Novice Programming Errors*. CA: Morgan Kaufmann, 1986.

[11] J. Anderson and B. Reiser. The lisp tutor. *Byte*, 10, 159-175, 1985.

[12] A. Mitrovic and S. Ohlsson. Evaluation of a constraint-based tutor for a database language. *International Journal of Artificial Intelligence in Education,* 10, 238-256, 1999.

[13] K. Koedinger, J. Anderson, W. Hadley, M. Mark. Intelligent tutoring goes to school in the big city. *Int. J. of Artificial Intelligence in Education*, 8, 30-43, 1997.

[14] B. Martin. *Intelligent Tutoring Systems: The practical Implementation of Constraint-based Modelling.* PhD thesis, University of Canterbury, 2001.

[15] A. Mitrovic, K. Koedinger, B. Martin. A comparative analysis of cognitive tutoring and constraint-based model-ling. *Proc. 9th Int. Conf. on User Modeling,* 313-322, 2003.

[16] K. Warendorf, C. Tan. Constraint-based student modeling - a simpler way of revising student errors. *Proc. ICICS,* 2, 1083-1087, 1997.

[17] P. Brna. *Prolog Programming, A First Course*. 2001.

[18] A. Bowles, P. Brna. Introductory Prolog: A suitable selection of programming techniques. In P. Brna, B.D. Boulay, H. Pain, *Learning to build and comprehend complex information structures: Prolog as a case study*. 167-177. Ablex Publishing Corp., Stamford, Connecticut, 1999.

[19] T. S. Gegg-Harrison. Learning prolog in a schema-based environment. *Instr. Science 20*, 173-192, 1991.

[20] J. Hong. Guided programming and automated error analysis in an intelligent Prolog tutor. *Int. J. Human-Computer Studies 61*, 505-534, 2004.

[21] M. Bieliková, P. Návrat. Learning programming in prolog using schemata. *ACM SIGPLAN Notices,* 33 (2):41-47, 1998.

[22] P. Brna, A. Bundy, T. Dodd, M. Eisenstadt, C. Looi, H. Pain, D. Robertson, B. Smith, M. Someren. Prolog program-ming techniques. *Instructional Science*, 20, 111-133, 1991.

[23] L. Sterling, M. Kirschenbaum. Applying techniques to skeletons. *Workshop on Constr. of L. Pro,* 1993.

[24] R.C. Sison, M. Numao, M. Shimura. Multistrategy Discovery and Detection of Novice Programmer Errors. *Machine Learning*, 38, 157-180, 2000.

[25] N.T. Le. Evaluation of a Constraint-based Error Diagnosis for Logic Programming. Proc.of the 13th Int. Conference on Computers in Education, 2005.