Using Prolog Design Patterns to Support Constraint-Based Error Diagnosis in Logic Programming

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

Abstract. Logic programming provides many possibilities to implement a task. Solutions can be realized by applying a variety of different design strategies and programming techniques. Using constraint-based modeling (CBM) we are in a position to cover the solution space for a given programming task. In this paper, we investigate the CBM approach for diagnostic purposes in the domain of logic programming. We address the ill-structuredness of this domain and the complexity of constraints modelling variations of design strategies. We propose an approach to structure the domain of logic programming by using Prolog patterns and to relieve the complexity of constraints by hypothesizing the design strategy in the student's solution.

Keywords: diagnosis, constraint-based modeling, logic programming, programming techniques, Prolog patterns.

INTRODUCTION

Prolog is one of the most widely used logic programming language. Prolog is considered to be difficult to learn because of the simple syntax and the concept of recursive programming which is the most important programming technique (Taylor 1999; Taylor & Boulay 1987). In addition, the domain of logic programming is infinite. For a given programming task, there is no single solution, but many strategies to design a solution. For a strategy, there are many ways to implement it. This causes students to search for a correct program for the given task.

Over the last two decades, numerous error diagnosis approaches in the domain of programming languages have been devised, such as program transformation (Vanneste, 1994; Xu and Chee, 2003), program verification (Murray, 1988), plan and bug library (Weber, 1996), model tracing (Anderson and Reiser, 1985) and constraint-based modeling (Ohlsson, 1994; Mitrovic et al, 2001). Among these, model tracing is used by cognitive tutors which are some of the most successful ITS today (Koedinger, Anderson, Hadley & Mark 1997).

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. By tracing the actions of the student with a collection of these rules, model tracing systems are able to return feedback immediately, whenever students perform a "bad" action. Model tracing approach has been applied to build a tutor system for Lisp (Anderson&Reiser 1985). However, the model tracing tutor is unable to trace the student's input when a student follows an unexpected but correct strategy.

While the model tracing technique has been widely applied for developing cognitive-motivated tutor systems (Martin 2001), recently, the constraint-based modeling (CBM) approach has been showing great promise as a diagnostic approach (Mitrovic et. al., 2001) which focuses on static cognitive states rather than problem solving processes. This approach has been employed successfully to build an SQL tutor system (Mitrovic & Ohlsson, 1999), natural language (Menzel, 2005), in the domain of data structures (Warendorf & Tan 1997) and has also been researched in the domain of UML (Baraghei, 2005).

In this paper, we investigate the CBM approach for diagnostic purposes in the domain of logic programming. We address the ill-structuredness of this domain and the complexity of constraints modelling variations of design strategies. We propose an approach to structure the domain of logic programming by using Prolog patterns and to relieve the complexity of constraints by hypothesizing the design strategy in the student's solution.

LOGIC PROGRAMMING

We intend to support students of Computer Science doing homework of logic programming course by using our tutoring system. The system provides students with task assignments and requests them to submit their program for the given task in a free form. A logic program consists of the definition of a main predicate and several auxiliary predicates which are necessary to solve sub-problems in the main predicate. A predicate definition is comprised of clauses and each clause has a clause head and several goals as the predicate `reverse/2` in Figure 1 illustrates. One of the main execution techniques of logic programs is unification. Unification is the way Prolog

does its matching. Two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal. (Blackburn, Bos and Striegnitz, 2001). Co-reference relationship is referred as a special case of unification which applies for arguments and functors.

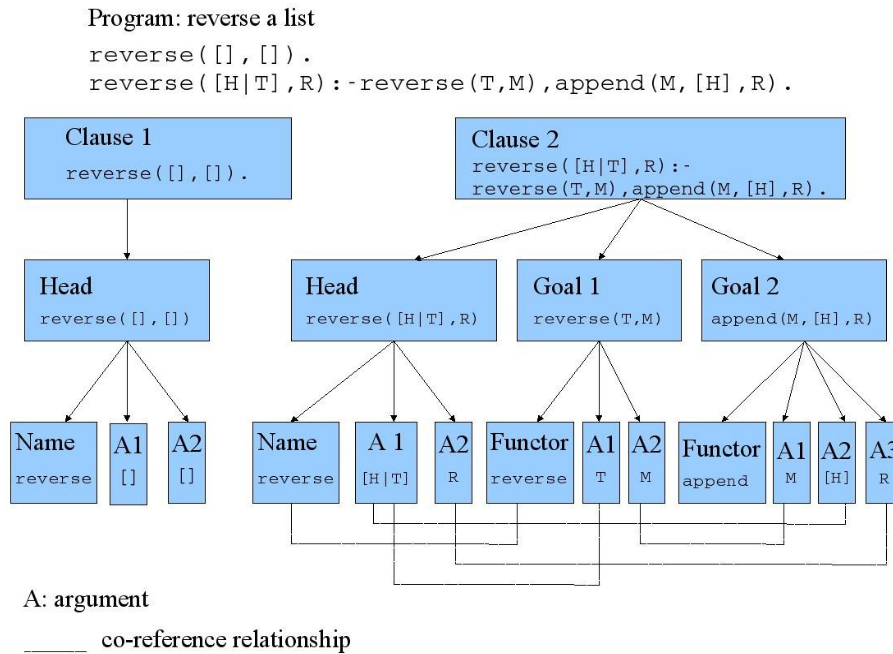

Figure 1 Structure of a Prolog predicate definition

Logic programming is an open-ended domain. The solution space for a programming task is infinite. There are three cases where solutions can be varied:

1. Different algorithms and design strategies provide various implementations. For instance, for the task of reversing a list we can apply different design strategies among which the common ones are e.g. naive recursion, inverse naive, railway-shunt and accumulator. Those design strategies are explained later on in this paper. For different strategies and algorithms, there are different implementations as the following example shows.

Naive	*Inverse naive*

`reverse([],[]).`
`reverse([H|T],R):-`
`        reverse(T,M), append(M,[H],R).`
	`reverse([],[]).`
	`reverse(L,[H|T]):-`
	`        append(M,[H],L), reverse(M,T).`

2. Different programming techniques can be applied for the same purpose. For instance, to realize a unification, we can choose between implicit unification and explicit unification. To implement explicit unification different operators can be applied. In the following clause, the second argument of the clause head unifies with the third argument of the call "`append`" implicitly:
```
reverse([H|T],R):-reverse(T,M),append(M,[H],R).
```
We also can implement that unification explicitly using **`R1`** and **`R2`** for different arguments and the unification between them is coerced by the operator `"="`.
```
reverse([H|T],R1):-reverse(T,M), append(M,[H],R2),R1=R2.
```

3. Auxiliary predicates can be defined according to individual needs. There are two reasons to define an auxiliary predicate:
    1. To execute a necessary subtask and enable the re-usability. The *Naive* definition for `reverse/2` above needs the auxiliary predicate `append/3` in order to concatenate an element to the end of a list. For the purpose of composition an element to the end of a list, one can use `append/3` or define an auxiliary predicate `enqueue/3` which queues a single element at the back of a list. Although the our system provides students the possibility to select appropriate built-in predicates like `append/3`, sometimes students still want to define their own auxiliary predicate like in the following *Naive* definition for `reverse/2`:
    ```
    reverse([],[]).
    reverse([H|T],R):-reverse(T,M), enqueue(H,M,R).
    enqueue(X,[],[X]).
    ```

```
enqueue(X,[H|T],[H|R]):-enqueue(X,T,R).
```

2. To keep the code in the main predicate definition simple. For example, the task of determining a list of persons whose age is greater than 18 can be implemented in `adult/2` with or without using an auxiliary predicate, where each element of the person list is a pair of name and age. The test subgoal A>=18 in Solution 1 is replaced by the call of an auxiliary predicate `greater18/1` in Solution 2.

Solution 1:	*Solution 2: using auxiliary predicate*

`adult([],[]).`
`adult([(N,A)|T], [(N,A)|R]):-`
`A>=18, adult(T,R).`
`adult([(N,A)|T], R):-adult(T,R).`
	`adult([],[]).`
	`adult([(N,A)|T], [(N,A)|R]):-`
	`greater18(A), adult(T,R).`
	`adult([(N,A)|T], R):-adult(T,R).`
	
	`greater18(X):-X >= 18.`

For a programming task, the number of applicable algorithms and programming techniques is limited. However, the variants of design strategies are numerous and the space of auxiliary predicates which might be defined by each individual is not predictable. Thus, the domain of logic programming is ill-defined. We apply the CBM approach to model the domain of logic programming and specify Prolog patterns to structure this domain.

CONSTRAINT-BASED MODELING

The CBM approach is proposed in (Ohlsson, 1994) 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>
```

where the relevance part represents circumstances under which the constraint applies, and the satisfaction part represents a condition that requires to be met for the constraint to be satisfied.

A constraint is used to describe a fact, a principle or a condition which must hold for every solution contributed by the student. For example, the following principle can be formulated as a constraint:

> <u>Example 1: a Prolog principle</u> an arithmetic expression of a logic program, e.g. "A is X+Y" can only be evaluated if all variables of the right hand side are bound.
>
> Constraint 1:
> Relevance: X is a variable of the right hand side of an arithmetic expression,
> Satisfaction: X ought to be bound.

Constraints are not only used to circumscribe facts, principles or conditions of a domain, they can also be used to specify the requirements of a task or to handle solution variations. Using the relevance part, constraints can be tailored according to an ideal solution, which represents the requirements of the given task. Ideal solutions enables us to check whether the student has answered the problem correctly, looking at the semantics. Additional requirements, which have to be satisfied in that specific situation, can be specified in the satisfaction part. We take an example from (Suraweera, Mitrovic and Martin, 2005) and specify the constraint 2 to examine the operator required in the task.

> <u>Example 2: a task requirement</u>
> "When I went to the shop to buy two loafs of bread, I gave the shopkeeper a $5 note and he gave me $1 as change. Write an expression to find the price of a loaf of bread using x to represent the price". It can be represented as 2x + 1 = 5 or 2x = 5 - 1.
>
> Constraint 2:
> Relevance: the left hand side of the ideal solution has a **+**,
> Satisfaction: in the student's solution, the left hand side has a **+** OR the right hand side has a **-**

If a constraint is violated, it indicates that the student solution does not hold principles of a domain or it does not meet the requirements of the given task. In order to be able to evaluate constraints, we need to define a formal representation for constraints.

Constraint's formal representation

We define a constraint as a tuple: (Type, Relevance, Satisfaction, Severity, Position, Hint).

- *Type* is "pattern", "exercise" or "general". Constraint types are used to control the diagnosis process;
- *Relevance* is a relevance part;
- Satisfaction is a satisfaction part;

- *Severity* indicates the severity of the constraint, it ranges between zero if the constraint is very important and one if the constraint is just informative;
- *Position* is the position of the structural elements which are considered in the relevance part. Position is only instantiated when the constraint is relevant and is used to indicate the error location in the student's solution in case the satisfaction part is not fulfilled;
- *Hint* is an instructional message explaining a principle of the domain or a requirement of the given task.

Syntactically, the relevance part and the satisfaction part are logical expressions, i.e. conjunctions of propositions about a problem state (Ohlsson, 1993). However, a problem state cannot be identified directly from a (partial) structure of a Prolog program due to two reasons:

First, the structural elements (clauses, goals, arguments, and functors) of a Prolog program are related to each other not only horizontally but also vertically (Figure 1). A Prolog program can be parsed of three levels: clause level, goal level and argument/functor level. From the horizontal view, the clause order and the goal order determine the relationship between clauses and goals, respectively. The unification and the argument order determine the relationship between arguments themselves and between them and functors. From the vertical view, the existence of an argument within a goal and the existence of a goal within a clause indicate the relationship between an argument and a goal, between a goal and a clause, respectively. The relationship between structural elements of a Prolog program is so complex that a partial structure of a Prolog program can not reflect a problem state sufficiently.

Second, more information can be detected, e.g. the instantiation state of an argument or the type of an argument, if a structural element is observed on a whole picture of its corresponding program. The instantiation state of an argument is obtained by using the predicate declaration and inferring instantiation states for all arguments from the begin of a clause to the occurrence of the argument being considered. Such information cannot be read off if only a partial structure of a Prolog program is considered.

As a result, we need to extract information about horizontal and vertical relationships between structural elements from a given Prolog program and a given predicate declaration in order to create three types of facts to serve the relevance part and satisfaction part:

headargument(Creator, ClauseIndex, ClauseType, AuxiliaryPredicateList, HeadName, HeadLength, ArgumentIndex, ArgumentType, ArgumentValue)[1]

bodyargument(Creator, ClauseIndex, SubgoalIndex, Functor, SubgoalType, SubgoalLength, ArgumentIndex, ArgumentType, ArgumentValue)

argumentmode(Creator, ClauseIndex, SubgoalIndex, ArgumentIndex, Value, InstantiationState)

The facts *headargument* and *bodyargument* contain information about each argument in the clause head and in the clause body, respectively. If a fact of type *argumentmode* exists, it expresses that the argument is bound, after its corresponding subgoal has been executed.

Relevance parts and satisfaction parts can be specified as conjunctions of the facts of *headargument, bodyargument* and *argumentmode*. If additional functions are necessary for constraint evaluation, they can be added into the specification of relevance parts and satisfaction parts. The constraint evaluation is carried out as follows: First, the relevance part of the constraint is matched against the facts extracted from a Prolog program. If there is a match, i.e. the constraint is relevant to the program, and then the satisfaction part is matched against the facts. If the satisfaction part is fulfilled, then the Prolog program is considered to be correct regarding to that constraint. Otherwise, it indicates a shortcoming in the program and the corresponding information will be returned for instructional purposes: the position of the structural element considered in the relevance part, the constraint severity and the hint encoded in the constraint.

Applying the CBM for Prolog

We categorize constraints into two classes: Prolog general constraints and exercise specific constraints. The former class represents general principles and conditions of Prolog. The constraint 1 mentioned above is an example of this class. The second class considers the specific requirements of the given task. Constraint 2 above is an instance of this type.

Each exercise specific constraint requires an ideal solution which encapsulates the requirements of a task. We extract facts (*headargument, bodyargument, argumentmode*) from ideal solutions to specify constraints. The information we want to attain from ideal solutions is: What kind of elements exist? What kind of relationship exists between different elements? In logic programming, for a design strategy, there are many various implementations. Therefore, it is necessary to define a canonical normal form for ideal solutions. Using ideal

[1] Creator: the program being considered is created by the student or the human tutor; ClauseIndex: the index of the clause within the program; ClauseType: the clause being considered is a base case or a recursive case; AuxiliaryPredicateList: a list of predicates which are used for the current clause; HeadName: the name of the clause head; HeadLength: the number of arguments the clause head consists of; ArgumentIndex: the index of the argument within the clause head; ArgumentType: an argument can have one of the types: variable, anonymous variable, list, atom, number, peano number, arithmetic or arbitrary; ArgumentValue: the structural representation of the argument itself; SubgoalIndex: the index of the subgoal within the clause body, beginning with number one; Functor: the name of the subgoal's functor; SubgoalType: a subgoal can have one of the types: arithmetic test, calculation, list manipulation, term test, user defined, relation, recursion or unknown; SubgoalLength: the number of arguments the subgoal consists of; InstantiationState: an argument is bound or free.

solutions, we can specify constraints which are in a position to cover solution variations. In Prolog, there are four levels of solution variations. 1) variation of operations over arguments, e.g. unification can be used implicitly or explicitly or a term comparison can be expressed using: either `X<Y` or `Y>X` or `not(X>=Y)`; 2) variation of the subgoal order, e.g. two subgoals can be transposed without changing the correctness of a program; 3) variation of the clause order, e.g. the order of two clauses can be changed while the semantics is preserved; 4) variation of implementation strategies for the same programming problem.

The following constraint which is able to cover different unification implementations for list arguments in a base case is a constraint of the first variation level.

Constraint 3:
Relevance:
the predicate of the ideal solution is a single recursion AND
there exists 1 base case AND
its clause head contains 2 arguments [H|R] at position Pos1 AND [H|T] at position Pos2 AND
the predicate of the student's solution is a single recursion AND
there exists 1 base case AND
its clause head contains 2 arguments [X1|Y1] at position Pos1 AND [X2|Y2] at position Pos2
Satisfaction:
in the student's solution the value of X1 must be the same as the value of X2 OR
there must exist a subgoal "X1 = X2" or "X2 = X1" in the body of the base case.

Similarly, we can specify constraints to cover solution variations of the subgoal level and the clause level. However, the CBM seems to encounter limitations to handle solution variations on the design strategy level. Exercise specific constraints modelling solution variations on this level are very complex. If the specification of the constraint is too complex, then it might happen that due to a slightly deviation the constraint will no longer be relevant to that solution.

The complexity problem

From Constraint 2 and Constraint 3 we derive a general formula for exercise specific constraints which should cover different implementations:

Formula 1:
$C_r(IS, SS, relevantobject) \rightarrow SS = C_s(IS) \vee SS = variant(C_s(IS))$

where the left hand side and the right hand side of the → represent the relevance part and the satisfaction part, respectively[2]. $C_r(IS, SS, object)$ refers to the problem state which is described by the `relevantobject` in the ideal solution and the student's program. The `relevantobject` can be an operation, a goal, a clause or a design strategy. $SS = C_s(IS) \vee SS = variant(C_s(IS))$ means the student's solution matches the state required in the ideal solution or a variant of that state.

Deriving from Formula 1 we define Formula 2 for constraints which cover solution variations of the design strategy level.

Formula 2:
$C_r(IS,SS,strategy) \rightarrow SS=C_s(IS,strategy) \vee SS=variant(C_s(IS,strategy))$

At this point, several questions will arise: 1) what are the elements which characterize a design strategy such that it can be distinguished from another one? 2) How can we specify the relevant part and the satisfaction part for a constraint which should cover different design strategies? 3) Assuming, we are able to specify such constraints, will they be relevant to a slightly erroneous solution, which is intended to be implemented following that strategy? In order to address these questions, we should have a look at the *Naive* implementation and *Inverse* naive implementation of the task `reverse/2` which reverses a list.

The first question can be answered by deriving from the *Naive* implementation of `reverse/2` the following description: a base case exists, a recursive case exists, the input list is decomposed in a head and a tail and the tail is decomposed recursively. The characteristic of the *Inverse naive* implementation can be described as follows: a base case exists, a recursive case exists, the input list is decomposed into a front list and a last element, the front list is decomposed recursively.

The second question can be answered by applying the descriptions above to specify the relevance part and the satisfaction part of an exercise specific constraint. The relevance part describes the characteristic of a design strategy and the satisfaction part describes additional requirements. The specification for such a constraint will be obviously very complex as the following Constraints 4 and 5 show, where **Naive** and **Inverse** are abbreviations of the description for the *Naive* strategy and the *Inverse naive* strategy above[3].

---

[2] the operator → is not a symbol for logical implication. Constraints are not inference rules (Ohlsson, 1993).

[3] The description for a design strategy is too long, thus we use a name to avoid repetition.

Constraint 4:
  Relevance:
    The ideal solution is **Naive** AND
    in the recursive clause, the composition subgoal is a built-in predicate "append" AND
    the student's solution is **Naive**
  Satisfaction:
    the student's solution ought to have a composition subgoal "append"

Constraint 5:
  Relevance:
    The ideal solution is **Naive**
  Satisfaction:
    The student's solution is **Naive** OR **Inverse**

Constraint 4 examines the application of a composition subgoal and Constraint 5 makes sure that the student's solution is implemented corresponding to either *Naive* or *Inverse***.**

Before we answer the third question, we investigate two cases:

Case A: if there is a student's solution which is intended to implement the *Naive* strategy but it misses a base case, then Constraint 4 will be not relevant to the student's solution, i.e. the student's solution is considered to be correct. This diagnostic information is too vague.

Case B: if there is a student's solution which is intended to implement the *Naive* strategy but it is erroneous, Constraint 5 will be violated as expected. However, due to the nature of that constraint specification, the location of the error can not be identified.

The two cases above answer the third question negatively and support our claim that constraints specified to cover solution variations on the design strategy level are not useful for diagnostic purposes. This is attributed to the high complexity of such constraints which are specified with a lot of information so that a slight error in the student's solution might cause the relevance part or the satisfaction part easily to evaluate to false. We propose to relieve the complexity problem by breaking down such constraints into two parts: a set of constraints which describe the characteristic of a design strategy and another set of constraints which examine additional requirements of that strategy. The former one is used to hypothesize the intention hidden in the student's solution.

Prolog patterns and design strategy hypothesis

Investigating the *Naive* implementations for different tasks, we notice that they share the same pattern. For instance, the following definitions for `insertion_sort/2` and `reverse/2` have the programming techniques in common: a base case exists, in the recursive case the input list is decomposed into a head and a tail, get the first element of the input list for processing, decompose the tail recursively and finally compose the result argument.

```
insertion_sort([],[]).
insertion_sort([H|T],R):- insertion_sort(T,S), insert(S,H,R).

reverse([],[]).
reverse([H|T],R):- reverse(T,M), append(M,[H],R).
```

Prolog programming techniques are programming practices which can be found in different contexts (Brna et al, 1999). A programming technique is language dependent, but specification 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 parts of a complete procedure, and many techniques may be combined together in a procedure. (Bowles & Brna, 1999) propose five essential Prolog programming techniques. The *same* technique is used to pass the same value between the head and the recursive subgoal of a clause. The *list head* is used when the head value is the list and the value of the recursive subgoal is its tail. The *list subgoal* technique is where the value of the recursive subgoal is the list and the head value is its tail. The *after* technique is used to indicate non recursive subgoals after the recursive one. The *before* technique is the opposite of the *after* technique. Techniques can be combined to create a new technique.

A design strategy in logic programming can be represented by a set of programming techniques which can be referred to as a Prolog pattern. We can apply the CBM approach to model Prolog patterns. The *Naive* pattern can be modelled by the following set of constraints using the formal representation of constraint(Type, Relevance, Satisfaction, Severity, Position, Hint)[4]. The constraints which are associated to a Prolog pattern are referred to as pattern constraints.

Constraint N1: *constraint(pattern, "patternname=naive", "a base case exists", 0.1, [], Hint1)*

Constraint N2: *constraint(pattern, "patternname=naive", "a recursive case exists", 0.1, [], Hint2)*

[4] For the simplicity we do not specify the error position and instructional hints in our constraint examples.

Constraint N3: *constraint(pattern, "patternname=naive and a recursive case exists", "the input list is decomposed in a head and a tail", 0.3, [], Hint3)*

Constraint N4: *constraint(pattern, "patternname=naive and a recursive case exists", "the tail is decomposed recursively", 0.3, [], Hint4)*

Constraint N5: *constraint(pattern, "patternname=naive and a recursive case exists", "the result argument is composed by the head and the process result of the tail", 0.3, [], Hint5)*

We specify the severity for constraints N1 and N2 as a value of 0.1 and for constraints N3, N4, N5 as a value of 0.3 because the constraints N1 and N2 are more important than the other ones. In addition, constraints N3, N4 and N5 assume that N2 evaluates to true. The severity is used to hypothesize the Prolog pattern implemented in the student's solution. The following formula determines how plausible the student's solution is realized corresponding to that Prolog pattern where `S1, S2...` and `Sn` are the severity of constraints which are violated if they are relevant to the student's solution.

Formula 3:

```
Plausibility(solution, patternname) = S1 * S2 * … *Sn
```

If the task `reverse/2` can be implemented by applying different Prolog patterns: *Naive, Inverse naive, Accumulator, Railway shunt* (Hong, 2004)[5], then we have to evaluate all four sets of pattern constraints for any given student's solution and to compute the plausibility for each Prolog pattern. The Prolog pattern which has the highest plausibility is considered to be the most plausible one implemented in the student's solution.

By means of Prolog patterns we are in a position to hypothesize the strategy in the student's solution and to examine the correctness of structural elements at deeper levels, even if the student's solution is not in agreement with the strategy properly. E.g. the constraint 4 above can be rewritten as the following constraint:

Constraint 4A: Relevance: patternname = naive AND in the recursive clause of the ideal solution, the composition subgoal is a built-in predicate "append" Satisfaction: in the recursive clause of the student's solution should have a built-in predicate "append"

Prolog Patterns can be organized two dimensionally. On the horizontal dimension, Prolog patterns can be found according to classes of programming problems. In general we can have four classes of programming problems: 1) test if any element of an input collection satisfies the given property, 2) test if all elements of an input collection satisfy the given property, 3) process one element of the input collection which satisfies the given property and return a result, 4) process all elements of the input collection and returns a result (Brna, 2001).

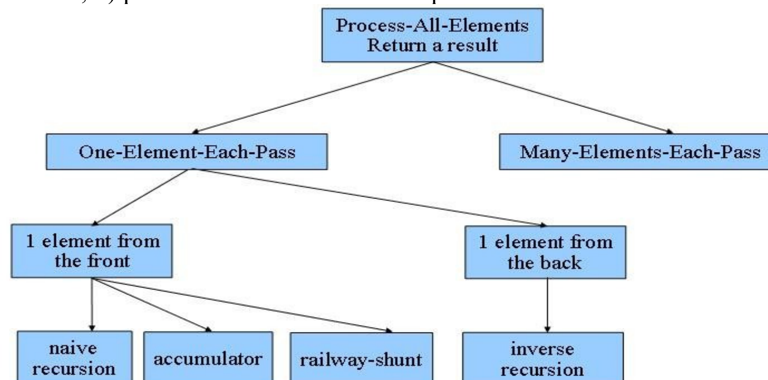

Figure 2 a small hierarchy of Prolog patterns

On the vertical dimension, Prolog patterns are differentiated by programming techniques. Figure 2 illustrates a hierarchy of Prolog patterns for the problem class of processing all elements of an input list and returning a result. The technique of list processing can be differentiated between the removal of one element or the removal of two elements from the input list for recursive processing. The technique of decomposition can be distinguished by taking an element from the front or from the back of a list. When an element has been decomposed from the front of a list, we can apply different programming techniques and have three patterns: *naive recursion, accumulator* or *railway-shunt*. If the decomposition takes place at the back of the input list, we apply the recursion on the element decomposed which specifies the *inverse* recursion pattern (Gegg-Harrison, 1993).

The benefits of Prolog patterns can be summarized in following points:

[5] Hong refers to the design strategy as a programming technique. We prefer the term Prolog pattern to avoid confusion with the notion of programming techniques in (Bowles & Brna, 1999).

1. Structure the domain of logic programming for diagnostic purpose. (see Figure 2)
2. Hypothesize the strategy implemented in a Prolog program. (see Formula 3)
3. Decrease the complexity of exercise specific constraints and thus, to increase the accuracy of the diagnostic information. (see Constraint 4A)

The problem of user-defined auxiliary predicates

If an auxiliary predicate is necessary to solve a subtask, then the ideal solution should also contain a corresponding definition of auxiliary predicates. The amount of possible design strategies to solve a subtask is countable.

However, if one wants to keep the code in the main predicate definition clear and simple, then he can define any auxiliary predicate and this is unpredictable. We need an ideal solution to encapsulate the exercise requirements to specify exercise-specific constraints. Which of the solutions should be the ideal one? Applying the unfolding/folding techniques proposed in (Tamaki & Sato, 1984) we are in a position to transform students solutions which uses auxiliary predicate to the one without auxiliary predicates, if not both the main predicate and the auxiliary predicate apply the recursion technique. The solution without auxiliary predicates is supposed to be the ideal one which has the normal form. If both the main predicate and the auxiliary predicate require recursion, then the ideal solution must contain the definition of the auxiliary predicate.

THE DIAGNOSIS PROCESS

Applying the CBM for Prolog we carry out the diagnosis in four steps:

1. transform the student's solution to the normal form
2. evaluate the pattern constraints
3. evaluate the exercise specific constraints
4. evaluate the Prolog general constraints

Violated constraints from the first step yield diagnostic information about the lack of the skill in designing a solution. Hints on this level are used to help students to master the application of Prolog patterns. The second step delivers information about application of programming techniques required for specific tasks. Diagnostic information from the third step reminds students to attend to the regularity of Prolog.

CONCLUSION AND FUTURE WORK

For a task in logic programming there is a variety of possible solutions. The amount of correct solutions for the given task is unpredictable. Thus, the task of logic programming is an ill-domain. We presented a way to structure this domain by using Prolog patterns and to model this domain applying the CBM approach. In addition, using Prolog patterns we are in a position to hypothesize the design strategy in the student's solution.

As a result, constraints are partitioned into different sets: pattern constraints, exercise-specific constraints and Prolog general constraints. For each constraint set, the instructional intention is differentiated. Pattern constraints are used to support students to master the activity of solution design by using a Prolog pattern. Both pattern constraints and exercise specific constraints can be encoded with instructional hints which help students to be familiar with programming techniques. The last set of constraints reminds students to consider the regularity of logic programming.

Until now we have implemented the evaluation component which is able to evaluate the three kinds of constraints mentioned above. We now need to verify the approach by establishing a constraint database and testing it on a wide range of programming tasks. We are also investigating the following problems:

- The problem of using user defined auxiliary predicates: folding/unfolding techniques are limited to certain types of solutions which use auxiliary predicates so that not both the main predicate and the auxiliary predicate apply recursion. For other solutions, which can not be transformed using folding/unfoldind, we need to invent another approach.
- The problem of task analysis: in many cases, students make errors because they have difficulties with task analysis. Thus, at that point, diagnostic information about semantic or syntactic errors is not relevant for students, but rather the stage of the problem solving process where the student becomes stuck in: problem analysis, solution design or implementation. We intend to develop an approach to determine how far the student has understood the task specification and to support students to analyse the given task.

REFERENCES

Anderson, J.R. and Reiser, B.J. (1985) *The Lisp Tutor*. BYTE, April, 159-175.

Baghaei, N. & Mitrovic, A. (2005) *COLLECT-UML: Supporting individual and collaborative learning of UML class diagrams in a constraint-based tutor*. Accepted for presentation at KES.

Blackburn, Bos and Striegnitz (2001) *Learn Prolog now*. http://www.coli.uni-sb.de/kris/learn-prolog-now

Bowles, A. & Brna, P. (1999) *Introductory Prolog: a suitable selection of programming techniques*. In: Brna, P., du Boulay, B., Pain, H.(eds), Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study. Ablex, pp. 167-178.

Brna, P. et al. (1999) *Programming techniques for Prolog*. In: Brna, P., du Boulay, B., Pain, H.(eds), Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study. Ablex, pp. 143-166.

Brna, P. (2001) *Prolog Programming, a First Course*.

Gegg-Harrison, T.S. (1993) *Exploiting program schemata in a Prolog tutoring system*. Ph.D. Dissertation, Technical Report CS-1993-11, Department of Computer Science. Duke University. Durham.

Hong, J. (2004) *Guided programming and automated error analysis in an intelligent Prolog tutor*. International Journal Human-Computer Studies 61, p.505-534.

Menzel, W. (2006) *Constraint-based modeling and ambiguity*. To be published in International Journal of Artificial Intelligence in Education, volume 16.

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

Mitrovic, A. et al. (2001) *Constraint-based tutors: a success story*. In L. Monostori and J. Vancza, Proceeding of the 14th Int. Conf. on Industrial Engineering Application of AI and Expert Systems, 931-940, Budapest.

Murray, W. (1988) *Automatic Program Debugging for Intelligent Tutoring Systems.* Los Altos, CA: Morgan Kaufmann, 1988.

Ohlsson, S. (1993) *The interaction between knowledge and practice in the acquisition of cognitive skills.* In Chipman, S. Foundations of knowledge acquisition.

Ohlsson, S. (1994) *Constraint-based student modeling.* In J. E. Greer, G.I. McCalla, Student Modelling: The Key to Individualized Knowledge-based Instruction, 167-189. Berlin.

Suraweera, P., Mitrovic, A., Martin, B. (2005) *A knowledge acquisition system for constraint-based Intelligent Tutoring Systems.* http://www.cosc.canterbury.ac.nz/tanja.mitrovic/Suraweera-AIED05.pdf

Tamaki, H. and Sato, T. (1984) *Unfold/Fold transformation of logic programs*. Proceedings of the 2nd International Logic Programming Conference, Uppsala, Sweden, 127-138.

Taylor, J. (1999) *Analysing novices analysing Prolog: what stories do novices tell themselves about Prolog*? In P. Brna, B. du Boulay, H. Pain (Eds). Learning to build and Comprehend Complex Information Structures: Prolog as a Case Study, Ablex, 43-71.

Taylor, J. and Boulay, B.D. (1987) *Studying novice programmers: why they might find learning Prolog hard.* In Rutkowska, J.C., Crook, C.(Eds), Computers, Cognition and Development: Issues for Psychology and Education. Wiley, New York.

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

Warendorf, K. and Tan, C. (1997) *Constraint-based student modeling - a simpler way of revising student errors.* In Proceedings of ICICS, 2, 1083-1087.

Weber, G. (1996) *Episodic learner modelling.* Cognitive Science, (20), 195-236.

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