Using Constraint-based Modelling to Describe the Solution Space of Ill-defined Problems in Logic Programming

Nguyen-Thinh Le, Wolfgang Menzel
University of Hamburg, Department of Informatics
Vogt-Kölln-Str. 30
D-22527 Hamburg, Germany
{le, menzel}@informatik.uni-hamburg.de

ABSTRACT
Intelligent Tutoring Systems have made great strides in recent years. Many of these gains have been achieved for well-defined problems. However, solving ill-defined problems is important because it can enhance the cognitive, metacognitive and argumentation skills of a student. In this paper, we demonstrate how to apply the constraint-based modelling approach to describe the solution space of ill-defined problems in logic programming. This technology has been integrated into a web-based ITS (INCOM) and has been evaluated with student solutions from past examinations.

Keywords
ill-defined problems, error diagnosis, logic programming, constraint-based modelling, ITS.

1. INTRODUCTION
Intelligent Tutoring Systems (ITS) have made great strides in recent years. Many of these gains have been achieved for well-defined problems such as geometry, Newtonian Mechanics, and system maintenance [10]. However, by solving ill-defined problems students can gain more benefits:

1.Enhancement of cognitive skills: well-developed domain knowledge is a primary factor in solving ill-defined problems [5, 14]. In solving ill-defined problems, students apply their domain knowledge in a meaningful way instead of storing a chunk of concepts in a memory [22].

2. Enhancement of metacognitive skills: ill-defined problems require solvers to control and regulate the selection and execution of a solution process [5, 2, 4]. In the ill-defined problem solving process, students employ their meta-cognitive skills, such as changing strategies, modifying plans and reevaluating goals in order to reach an optimal solution [22].

3. Enhancement of argumentation skills: since ill-defined problems require students to consider alternative solutions, successful students can provide evidence for their solution [19, 20]. Therefore, students gain practice justifying their solution in a logical way to persuade others.

For this purpose, we focus our research on a web-based programming ITS which supports students learning logic programming by solving ill-defined problems. Prolog is one of the most widely used logic programming languages. 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 [17]. In general, the domain of 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 them.

How can an ITS diagnose errors in the student solution for an ill-defined problem? Over the last two decades, numerous error diagnosis approaches in the domain of programming languages have been devised, such as program transformation [18, 23], program verification [11], plan and bug library [21], model tracing [1] and constraint-based modeling (CBM) [12]. Among these, model tracing is used by cognitive tutors which are some of the most successful ITS today [7]. However, those approaches have been applied to problems with a lower degree of ill-definedness. An ill-defined problem has not only a simple correct solution, rather many or even uncountably. In this paper, we introduce the CBM approach to cope with the solution space for ill-defined problems in logic programming.

In the next section, we review the characteristics of ill-definedness in the literature and argue why logic programming problems provided by INCOM are ill-defined. The solution space for a programming problem in Prolog is described in the third section. In the fourth section, we introduce CBM and how we apply it to model the solution space for a Prolog problem. The fifth section illustrates the architecture of INCOM briefly. In the sixth section we show our evaluation result. Our conclusions and future works are summarized in the last section.

2. ILL-DEFINED[1] PROBLEMS
In the literature, there is no formal definition what constitutes a “well-defined problem”. Instead, we must be

[1] In this paper we have chosen the term ill-defined problem. The terms ill-structured and ill-defined are used interchangeably in the literature. To avoid confusion, we will use the former one.

content with requirements which have been proposed as criteria a problem must satisfy in order to be regarded as well-defined: 1) a start state is available; 2) there exist a limited number of relatively easily formalized transformation rules; 3) evaluation functions are specified and 4) the goal state is unambiguous [6]. If one or several of these conditions is violated, the problem is considered ill-defined [13]. Most researchers agree that a design problem is a representative of ill-defined problems [3], because the start state is underspecified, there is no predefined set of rules for completing the task, and it is difficult to evaluate when a "best" result has been attained. However, "*the boundary between well-defined and ill-defined problems is vague, fluid and not susceptible to formalization*" [15]. Thus, this vagueness and relativity should simply reflect the continuum of degrees of definiteness between the well-defined and ill-defined ends of the problem spectrum. Those problems, which lie somewhere in the middle between well-defined and ill-defined ends, may have well-specified start and goal states, but underspecified transformation rules or evaluation functions because 1) there are multiple representations of knowledge with complex interactions; 2) the ways in which the rules apply vary across cases nominally of the same type [16] and 3) there are only aesthetic value judgments, but no quantitative measurements available.

Most programming problems, which are used to tutor Prolog, are simple and might have well-defined start as well as goal states. The problem text can be well specified and a solution can be verified whether it solves the given problem correctly. However, the activity of solving a Prolog programming problem is a design problem. The solution space is mainly spanned by using different Prolog primitives or applying mathematical rules. Furthermore, one can modularize a program in order to make the code clearer, easy maintainable and reusable, but all these criteria are highly subjective. That is why Prolog programming problems are ill-defined. We provide students with a series of problem tasks (Appendix A) to be solved in free-form using our system INCOM.

3. SOLUTION SPACE FOR A LOGIC PROGRAMMING PROBLEM

We consider only Prolog programs without cuts, disjunctions or if-then-else operators. No assert, retract, abolish or similar database-altering predicate can be used. The set of built-in predicates which can be employed by the programs are: =, =.=, =\=, ==, \==, >, >=, <, =<, =.., +, -, *, /, ^ and 'is'. Auxiliary predicates are provided explicitly or can be defined by users.

To solve a logic programming problem, there are many solutions. The solution space results from a variety of Prolog primitives and a variety of programming techniques which describe standard solution strategies. The solution space is also determined by a set of general principles of Prolog. Furthermore, it is restricted by an appropriate predicate declaration which is developed interactively with the student prior to the implementation itself. The declaration information is used to gather the intention of the student, and to understand subsequent implementation of the student solution. In the following, we describe the space of solutions for a given problem in more detail.

Solution space spanned by Prolog primitives: a predicate is composed of a clause head and a clause body. A clause body contains several subgoals. Table 1 shows the possible variations of a clause head and of its subgoals. This collection is not a complete one, but also reflects specific restrictions imposed by the system. The following types of clause head and subgoals can be distinguished:

Clause head: a clause head is the first part of a clause of a predicate. The definition for a clause head must adhere to the predicate declaration: clause type (a base case, a recursive case or a non-recursive clause), argument types (atom, number, list or arbitrary) and argument modes (+, -, ?). A clause head may vary depending on the clause body, i.e. a (de)composition or a unification either takes place in the clause head implicitly or in the clause body explicitly.

Recursive: a recursive subgoal has the same functor and the same arity as its clause head. Arguments of an recursive subgoal inherit declaration information from the clause head such as: types, modes and argument meaning.

(De)composition: a (de)composition subgoal composes an argument using other variables or decomposes an argument into several variables or constants. A (de)composition can be established implicitly at an argument position or can be represented explicitly as a separate subgoal.

Arithmetic test: an arithmetic test subgoal is used to compare two bound arguments which are of type number. There are two classes of arithmetic test subgoals. The first one applies the operators: <, >, =< and >= to test whether a number is greater/smaller than another one. The second class applies the operators: =:= and =\= to test whether two numbers are equal or not. The operands of the operators can be transposed because they are commutative.

Calculation: a calculation subgoal is used to compute an arithmetic expression using the operator 'is'. It requires that the variables on the right hand side of the subgoal are bound; otherwise the evaluation cannot be executed. We consider the following arithmetic operators: +, -, *, /, ^ and three forms for an arithmetic expression are distinguished: 1) Normal form: A°X±B°X; 2) Applying the distributive law and 3) Applying the commutative law where the operator ± is either + or -, and the operator ° is either * or /. Currently we do not perform any transformation for exponential expressions: (X+Y)^A. We just consider the neutral elements: X^1=X and X^0=1. The basis is regarded as an arithmetic expression with possible variants.

Unification: a unification subgoal unifies two variables or assigns a value to a variable using the operator =. The unification subgoal is referred to as an explicit unification.

A unification can also occur if two different argument positions have the same variable name and this case is called implicit unification or co-reference.

Term test: a term test subgoal is intended to test whether two terms are equivalent using the operators: == and \==. We also include the operator \= into the class of term test because it tests whether two terms are not unifiable. All three operators are commutative.

Relation: a subgoal is a relation if a database is provided in which an appropriate relation is defined. A relation can not be transformed.

Helper predicates: we restrict the space of helper predicates to the ones which build accumulation over lists or which are defined without using recursion. In general, the space of helper predicates is open-ended.

Table 1. Normal form and variation of clause heads/subgoals

Head/Subgoal	**Normal form**	**Variants**
Clause head	p(X,Y):-X=Y. q(X,Y):-X=[H|T].	p(X,X). q([H|T],Y).
Recursive	p([H|T],Y):-p(T,Y)	p([H|T],Y):-p(T,Y)
(De)composition	Explicit: p(X,Y):-X=[H|T], p(T,Y).	Implicit: p([H|T], Y) :- p(T,Y).
Arithmetic test	X<Y A=:=B A=\=B	Y>X; X-Y<0; Y-X>0; 0>X-Y; 0<Y-X; B=:=A B=\=A
Calculation	A°X±B°X	distributive: (A±B)°X commutative: X°A±B°X, A°X±X°B, X°(A±B)
Unification	Explicit: p([H1|T1],[H2|T2]):- H1=H2, p(T1,T2).	Implicit: p([H|T1],[H|T2]):-p(T1,T2).
Term test	A==B A\==B A\=B	B==A B\==A B\=A
Relation	query(A,B,C)	query(A,B,C)
Helper predicate	help(A,B,C)	help(A,B,C)

Solution space spanned by patterns: to solve a problem in logic programming many programming techniques can be applied and combined. The programming techniques underlying a predicate definition determine the strategy of a problem solution. For example, to solve the problem of processing all elements of a list, one can choose between recursion by processing many elements or only one element. If only one element is processed, there are several possible alternatives: naive recursion, inverse recursion, recursion using an accumulation predicate or applying the railway-shunt [8]. Hence, for a given problem of Prolog programming, there are typical solution strategies which we refer to as patterns. The number of patterns spans the solution space for a given problem in logic programming.

Solution space spanned by a set of general principles of Prolog: the general principles of Prolog assure that a Prolog predicate definition is executable. The following is a subset of general principles of Prolog, which is not an exclusive list:

- Variables on the RHS of an calculation subgoal must be bound.
- Variables of an arithmetic test and of term test subgoals must be bound.
- For a recursive implementation, at least a base case and a recursive case are required

Solution space spanned by the choice of names for variables and predicates: as we do not want to restrict students to a small space of solutions, they are allowed to choose any names for variables and predicates as they do without an ITS system. Therefore, the solution space for a programming problem in Prolog also becomes open with respect to the choice of names for variables and predicates.

4. APPLYING CBM APPROACH

4.1 Constraint-based Modeling

The CBM approach has been proposed in [12] 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 has to be met for the constraint to be satisfied. 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 the semantics which represent the requirements of the given task [8]. 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 evaluate constraints, we define a formal representation for constraints: *constraint(Id, Type, Relevance, Satisfaction, Severity, Position, Hint)*[2]. Information about relationships between structural elements of a given Prolog program and a given predicate declaration are stored in three types of assertions: *headarg, bodyarg* and *argmode* where *headarg* and *bodyarg* contain information about each argument in the clause head and in the clause body, respectively. If an assertion of type *argmode* exists, it denotes that the argument is bound, after its corresponding subgoal has

[2] *Id* is an unique identification of the constraint; *Type* is one of *pattern, general, head, recursive, arithmetictest, termtest, (de)composition, unify or calculation*; *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 important and one if it is informative; *Position* indicates the error location; *Hint* is an instructional message.

been executed [8]. As a result, the relevance and satisfaction parts of a constraint can be specified as conjunctions of assertions. The constraint evaluation is carried out as follows: first, the relevance part of the constraint is matched against a set of assertions. If there is a match, i.e. the constraint is relevant to the program, then the satisfaction part is matched against the same set of assertions. If the satisfaction part is also fulfilled, then the Prolog program is considered to be correct with respect to that constraint. Otherwise, it indicates a shortcoming in the program and the corresponding information will be returned for instructional purposes: error location, constraint severity and hint encoded in the constraint [9].

4.2 Applying CBM to Model a Solution Space

There are two classes of constraints: semantic constraints and Prolog general constraints. The first one includes constraints which examine whether a solution object satisfies the requirements of a given problem. Constraints of the latter class examine whether the solution object fulfills general principles of Prolog. Prolog general constraints can be constructed as demonstrated in [8]. Semantic constraints are constructed based on a semantic table which contains information required to solve a given problem. Clause heads and subgoals in the semantic table are represented in normal forms (Table 2). The normal form representation reveals the underlying programming techniques and thus, the diagnosis becomes more accurate.

Table 2. An example of a semantic table

Head	Subgoal	Description
salary(OldL,NewL)	OldL=[] NewL=[]	Old list is empty New list is empty
salary(OldL,NewL)	OldL=[N,S|T] NewL=[N,Snew|Tnew] S=<5000 Snew is S+S*0.03 salary(T,Tnew)	N, S: name, salary build a new salary list Salary less than 5000 Salary is increased Decompose old salary list recursively

From this table, *headarg, bodyarg* and *argmode* assertions can be extracted. They are referred to as semantic assertions, while similar ones derived from the student solution are referred to as student assertions. In order to construct a semantic constraint, two steps are required:

1. We map semantic assertions to student assertions of the same subgoal type. This might result in multiple combinations of maps. For example: the set of semantic assertions contains two assertions *bodyarg(tp,unify,1,A)* and *bodyarg(tp,unify,2,B),* which represent a unification subgoal of two arguments **A** and **B** at position 1 and position 2, respectively. Similarly, two student assertions *bodyarg(sp,unify,1,SB)* and *bodyarg(sp,unify,2,SA)* represent a unification subgoal of two arguments **SA** and **SB** at position 2 and 1, respectively. Mapping two semantic assertions against two student assertions of the subgoal type *unify* results in two mappings:

Mapping1=[map(bodyarg(tp,unify,1,A),bodyarg(sp,unify,2,SA)), map(bodyarg(tp,unify,1,B),bodyarg(sp,unify,1,SB))]

Mapping2=[map(bodyarg(tp,unify,1,A),bodyarg(tp,unify,1,SB)), map(bodyarg(tp,unify,2,B), bodyarg(sp,unify,2,SA))]

2. We select relevant semantic assertions which consider a semantic unit for the relevance part. The satisfaction part checks corresponding student assertions in the selected mapping. The following formula computes the plausibility of each mapping: *P(Assertions,Mapping)=P1*P2*...*Pn* where *P1, P2*, ... are the severity of constraints which are violated. The mapping (either *Mapping1* or *Mapping2*), whose evaluation obtains the best score, is the most plausible one. If the student solution deviates from the normal form, then the satisfaction part has to consider all possible variants. In this case, a semantic constraint can be generalized as follows:

General Constraint Template:

constraint(Id, Type, Facts, Relevance: $s \subset S$ *, Satisfaction: test(SP,variant(s)), Severity, Position, Hint)*

s is a subset of semantic assertions **S** and describes a semantic unit, for instance: an explicit unification; **SP** is a subset of student assertions and **test(SP, variants(s))** tests whether the student solution satisfies the selected semantic unit **s** or its variants. The generalized constraint template above can be applied to: 1) Clause level: constraints examine clause order; 2) Clause subgoal level: constraints examine subgoal order, functor and arity of subgoals; 3) Argument level: constraints examine co-references of variables, correctness of constants or operators.

The two steps mapping-evaluation above allows us to cover the solution space spanned by the choice of names for variables and predicates because the mapping step maps corresponding elements between student assertions and semantic assertions without considering variable names or predicate names, and the evaluation step examines whether the mapping satisfies semantic units.

4.3 Applying CBM and Transformation to Model a Solution Space

As the right hand side (RHS) of a calculation subgoal can be transformed using the distributive and commutative law, there are difficulties to apply the generalized constraint template above directly. In addition, the structure of a calculation subgoal must be decomposed according to its depth. Currently, we do not consider recursively embedded arithmetic expressions. Our system just copes with arithmetic expressions without nesting, for example: A*(B+C). Before we attempt to model a space of calculation subgoals using CBM, we introduce transformation rules for arithmetic expressions and an algorithm for evaluating a calculation subgoal of the student solution whose RHS is referred to as SP_RHS.

Rule 1: transforms the normal form to the simplified form applying the distributive law: A°X ± B°X → (A±B)°X, where the operator ° is either * or /. If A and B are numbers, then (A±B)°X can be transformed to M°X where M=A±B. For example: (2+3)*X → 5*X

Rule 2: transforms a product term applying the commutative law: A*B -> B*A

Evaluate a calculation subgoal:

1. Evaluate the left hand side of the subgoal;

2. Call the algorithm **Evaluate arithmetic expression** to evaluate the RHS of the subgoal

Evaluate an arithmetic expression: the RHS of a calculation subgoal in the semantic table is represented in the normal form which is referred to as SEM_NF. That is a sum of many summands. Each summand is a product of many factors which are connected by arithmetic operators: *, /, ^. The arithmetic expression A*X + 1/Y – Z^B, for instance, is in normal form where the summands are A*X, 1/Y and Z^B. In order to evaluate an arithmetic expression, we have to evaluate from the arithmetic expression level through the summand level to the factor level as follows:

1. Apply Rule 1 to SEM_NF producing the simplified form SEM_SF. For instance, SEM_NF=3*Y+5*Y is simplified to SEM_SF=8*Y.

2. Create mappings which contain maps between the SEM_NF and SP_RHS as well as between SEM_SF and SP_RHS, respectively. If SP_RHS=8*X, for instance, mapping between SEM_NF and SP_RHS yields *Sum_Mapping1=[map(sum,3*Y,8*X),map(sum,5*Y,nil)]; Sum_Mapping2=[map(sum,5*Y,8*X),map(sum,3*Y,nil)];* and mapping between SEM_SF and SP_RHS results in *Sum_Mapping3=[map(sum,8*Y, 8*X)]*.

3. Call the algorithm **Evaluate summands** to evaluate constraints based on those mappings. The mapping which yields a better evaluation result defines the most plausible representation of SP_RHS. Following the example above, we can hypothesize that SEM_SF is the most suitable representation of SP_RHS because *Sum_Mapping1* and *Sum_Mapping2* contain a map of a product and a nil-value which causes constraint violation.

Evaluate summands: a summand is comprised of an algebraic sign and a list of factors. For instance, a summand 8*X*Z^B has the algebraic sign + and the factors [8, X, Z^B]. We refer to a summand of SEM_NF as SEM_SUM and a summand of the SP_RHS as SP_SUM. The algorithm of evaluating summands follows:

1. Evaluate the algebraic sign of SP_SUM whether it corresponds to the one of SEM_SUM.

2. If any element in the factor list of SEM_SUM contains a division, then apply Rule 2 to create a list of commutative variants, otherwise this list contains just SEM_SUM. Similarly, apply Rule 2 to SP_SUM.

3. Create mappings between SEM_SUM and SP_SUM by selecting one commutative variant of SEM_SUM and one variant of SP_SUM, then map factors of SEM_SUM against factors of SP_SUM.

4. Call the algorithm **Evaluate factors.** The factor mapping, which yields the best evaluation result, indicates that the selected factors of SP_SUM corresponds to factors of SEM_SUM.

Evaluate factors:

1. Evaluate the existence of factors.

2. Evaluate the type of each factor (variable, number, division or an exponential term) and co-references of variables, correctness of numbers.

3. If a factor is an exponential term and its basis is an arithmetic term, then apply the algorithm **evaluate an arithmetic expression** to this arithmetic term.

Applying the generalized constraint template above, we are in a position to construct constraints for evaluating factors, for evaluating summands and for evaluating arithmetic expressions. As a result, constraints evaluating calculation subgoals are nested constraints. That is, the relevance part of a constraint includes the execution of the evaluation on the next level and the satisfaction part requires that no error occurs during that evaluation.

4.4 Applying CBM and Pattern Candidates to Model a Solution Space

The variation of a subgoal (clause head) requires considering arguments within that subgoal (clause head). A pattern variant differs from others not only at argument positions in one subgoal, but also in many subgoals. Therefore, the generalized constraint template above cannot be applied directly to define constraints which span the space of patterns. Syntactically, a constraint, which should cover the space of patterns, requires to consider many subgoals (clause heads) in its relevance part. The following problem exercise illustrates this issue:

Compound interest: *An amount of money S will be charged with an annual interest rate (i.e. R=0.05) and rises exponentially. Define a predicate to compute the amount of money after X years of investment.*

For the problem above we can apply four different patterns: *analytic, tail recursive, increasing recursive and decreasing recursive*. Possible solutions according to the last two patterns are shown in Table 3. We can notice that those solutions differ from each other remarkably. Even though the *increasing recursive* and the *decreasing recursive* solutions seem to have many structures in common, from the view of underlying programming techniques, they are quite different. The predicate in the *increasing recursive* solution calls itself until the variable *New_Period* is bound, then the second subgoal, an arithmetic calculation, tests the bound value of

New_Period in relation to the bound value X (number of investment years). If the test succeeds, the new sum is calculated, otherwise, the recursive subgoal is called again. In another solution *New_Period* is calculated by decrementing X, as long as X is greater than 1, the new investment sum is computed until *New_Period* is zero.

Table 3. Possible solutions for the problem *Compound interest*

Pattern	Solution

Increasing recursive	`in_inv(S,_,0,S).`
	`in_inv(S,R,X,End_S):-`
	`in_inv(S,R,New_Period,New_S),`
	`X is New_Period+1,`
	`End_S is New_S+R*New_S.`

Decreasing recursive	`de_inv(S,_,0,S).`
	`de_inv(S,R,X,End_S):-X>0,`
	`New_Period is X-1,`
	`de_inv(S,R,New_Period,New_S),`
	`End_S is New_S+R*New_S.`

Hypothesis: It is possible to define a constraint which models a common space of solutions for both patterns *increasing recursive* and *decreasing recursive.*

Attempt 1: we define Constraint P1 without using a transformation rule. We select assertions from the semantic table to describe the *decreasing recursive* pattern as the relevance part of a constraint, and the satisfaction part requires that the student solution should satisfy either the *increasing recursive* or the *decreasing recursive* pattern.

Constraint P1:

Relevance: in the semantic table, a recursive subgoal R, an increment calculation subgoal A: X is V+1, a calculation subgoal A, and a base case C_{base} exist and should fulfill the following conditions: R precedes A; V also exists in R at the position p(V); p(V) is equal to the argument position of X in the clause head; The argument at position p(V) in C_{base} is bound to a constant.

Satisfaction: in the student solution, either condition set A or condition set B should be satisfied:

Condition set A: there should exist a recursive subgoal SR, an increment calculation SA: SX is SV+1, and a base case $SC_{base,}$ which meet following requirements: SR precedes SA; SV exists in SR at the position p(SV); p(SV) is equal to the argument position of X in the clause head; The argument at position p(SV) in SC_{base} is bound to a constant.

Condition set B: there should exist a recursive subgoal SR, a decrement calculation subgoal SC: SV is SX-1, and a base case SC_{base}, which meet the following requirements: SR precedes SC; SV exists in SR at the position p(SV); p(SV) is equal to the argument position of SX in the clause head; The argument at position p(SV) in SC_{base} is bound to a constant; An arithmetic test subgoal SX>0.

Suppose, the solution SP3 (Appendix B) for the problem *Compound interest* should be evaluated. The relevance part of Constraint P1 is always evaluated to true because it is a conjunction of semantic assertions. However, SP3 will not satisfy Constraint P1 because the calculation subgoal neither satisfies the condition set A nor set B. As a result, Constraint P1 is not useful due to following problems: 1) The student solution is implemented according to either the *increasing recursive* or the *decreasing recursive* pattern. But in case of an erroneous solution, this distinction is not reflected in the diagnostic results because the constraint simply evaluates to false without indicating the strategy used by the student probably. 2) The relevance part of Constraint P1 is so complex that errors in the student solution can not be reliably localized.

Attempt 2: we define a constraint using a transformation rule which transforms the *decreasing recursive* (DRP) to the *increasing recursive* pattern (IRP).

Rule 3: copy the base case of DRP: `de_inv(S,_,0,S);` copy the clause head of the recursive case of DRP: `de_inv(S,R,X,End_S);` remove the arithmetic test subgoal of DRP; convert the decrement subgoal of DRP to increment subgoal: `N_Period is X-1` → `X is N_Period+1;` concatenate the recursive subgoal of DRP to the front of the increment subgoal; concatenate the increment subgoal with other subgoals of the recursive case of DRP; the new form is the IRP.

We can now evaluate Constraint P1 on a predicate which uses arithmetic recursion as follows: 1) Assuming, a solution is coded according to *decreasing recursive* pattern (called SEM_DRP). We apply Rule 3 to SEM_DRP resulting in a predicate SEM_IRP which uses *increasing recursive* pattern; 2) We evaluate Constraint P1 based on either SEM_DRP or SEM_IRP; 3) SEM_DRP or SEM_IRP, which causes fewer constraint violations, is taken as the one which apparently has been implemented in the student solution. However, the transformation is very complex and it is difficult to verify its correctness.

To avoid the constraint complexity as in Attempt 1 and the necessity to apply Rule 3 as in Attempt 2, the semantic table is extended to contain two candidates for the two patterns. The student solution is evaluated based on semantic assertions extracted from the semantic table. The pattern candidate, which causes the least constraint violation, is taken as the most plausible explanation for the student solution.

5. IMPLEMENTATION

The architecture of our web-based INCOM is comprised of three layers: front-end, back-end and resource layer. The front-end layer plays the role of presenting exercise tasks to students, reading student solution inputs and returning feedback. The back-end layer is charged to transform student solutions to other possible variants, to analyse its structure and to diagnose errors by calling the General Constraint Evaluator. The resource layer contains exercise descriptions and associated semantic tables. The resource and back-end layers are implemented using SWI-Prolog, whereas the front-end layer is implemented using

JavaBeans and Java Server Pages. Front-end and back-end layer communicate via the Tomcat server.

6. EVALUATION

The efficacy of an ITS depends on the accuracy of diagnosis. We conducted an off-line test by selecting appropriate exercises from past written examinations and integrated them into INCOM. Then, we collected student solutions for those exercises. The examination candidates should have attended a course in logic programming which was offered as a part of the first semester curriculum in Informatics. The purpose of the evaluation is to find out whether the solution space modeled by CBM covers possible student solutions, and thus the ITS is in a position to give appropriate diagnostic information.

Currently, we conducted the evaluation with three exercises tasks (Appendix A). Each of the exercise tasks requires different skills to solve. For the first one, students should be able to handle recursion, arithmetic calculation, arithmetic test, and (de)composition of a list structure. The second one requires students to cope with arithmetic calculation and database relationships. The last one requires the skill of implementing a recursive subgoal, using unification and (de)composition.

While collecting student solutions from past examinations we filtered out solutions which are not sufficiently elaborated for applying a diagnosis, i.e. fragmentary clauses. In addition, we added appropriate predicate declarations, because students were not asked to provide that information about meaning, types and modes of each argument position. An expert marks the position of errors in the student solution, and looks for a list of possible constraints which might be violated. Finally, we run the diagnosis on the student solution resulting in a list of constraint violation hypotheses. If both lists are in agreement, the automatic diagnosis is in accordance with the one of the expert. Each student solution is a test case.

Table 4. Evaluation of student solutions

Exercise	Participants	Solutions	Solutions not diagnosed
1	20	11	0
2	20	5	2
3	39	10	0

We summarize the number of participants who had to solve the exercises and the number of collected solutions which are diagnosable in Table 4. The system INCOM could diagnose almost all collected solutions correctly except two of them: SP1, SP2 (Appendix B). In one case the solution contained a disjunction operator ';' which is currently not supported. In the other case the semantic table was incomplete, since it did not contain a pattern candidate implementing an accumulator.

7. CONCLUSION AND FUTURE WORKS

We have discussed how the CBM technique can be applied to describe the solution space of ill-defined problems in logic programming. There are three cases when we should apply constraints: 1) Constraints without transformation can be used to describe an object for which there is a small number of variants as long as the conditions are not too complex. If the complexity of a constraint becomes too high, the author risks that the relevance part of the constraint will not be fulfilled by an erroneous solution or that the error committed by the student cannot be diagnosed precisely enough. Therefore, we should use constraints without transformation primarily to examine objects at the argument level of a Prolog predicate definition. 2) Constraints with transformation can be applied to objects, which may have many variants, i.e. an arithmetic expression that can be modified using the distributive or commutative law. We suggest to describe only transformation rules which are verifiable. 3) Pattern candidates are required if we are not able to define a transformation between them or a transformation rule can not be verified. Normally, pattern candidates have a large degree of dissimilarity. They are distinguished from each other not only at the argument level but also at the subgoal and clause level and thus, a transformation becomes very complex. As a consequence, it is almost not possible to verify the correctness and the generality of the transformation. Therefore, a normalized pattern candidate should be provided.

This technology has been integrated into a web-based ITS and the system has been evaluated with student solutions from past examinations. The evaluation results pointed out that the CBM was able to cover 24 of 26 student solutions for the exercises in Appendix A. The disadvantage of this approach is that we have to define enough pattern candidates to represent the different programming strategies for which no transformation rules between patterns can be defined and verified. This is not always an easy task for a very ill-defined problem. We will be extending INCOM with new exercises and test cases to demonstrate the effectiveness of the CBM approach.

8. REFERENCES

[1] Anderson, J. R. & Reiser, B. J. *The Lisp Tutor*, BYTE, April, pp. 159-175, 1985.

[2] Gick, M. L. *Problem-solving strategies*, Educational Psychologist, Vol. 21, pp-99-120, 1986.

[3] Goel, V. *Comparison of well-structured & ill-structured task environments and problem spaces*, Proceedings of the 14th annual conference of the cognitive science society, Hillsdale, NJ: Erlbaum, 1992.

[4] Jacobs, J. E. & Paris, S. G. *Children's metacognition about reading: Issues in definition, measurement, and instruction*, Edu. Psychologist, 22, pp. 255-278, 1987.

[5] Jonassen, D. H. *Instructional design models for well-structured and ill-structured problem-solving learning*

outcomes, Educational Technology: Research and Development, Vol. 45, No. 2, pp. 65-94, 1997.

[6] Jonassen, D.H.; Tessmer, M. & Hannum, W.H. *Task analysis methods for instructional design*, Erlbaum 1999.

[7] Koedinger, K. R.; Anderson, J. R.; Hadley, W. H. & Mark, M. *Intelligent tutoring goes to school in the big city*, International Journal of Artificial Intelligence in Education, Vol. 8, No. 1, pp. 30-43, 1997.

[8] Le, Nguyen-Thinh *Using prolog design patterns to support constraint-based error diagnosis in logic programming*, in K. Ashley, V. Aleven, N. Pinkwart and C. Lynch (Ed.), Proc. of the Workshop on ITS for Ill-Defined Domains, the 8th Conf. on ITS, pp. 38-46, 2006.

[9] Le, Nguyen-Thinh, *INCOM: a constraint-based tutoring system for logic programming.* Report, FBI-HH-B-280/07, University of Hamburg, Department of Informatics.

[10] Lynch, C. F.; Ashley, K. D.; Aleven, V. & Pinkwart, N. *Defining Ill-Defined Domains; A literature survey*, In Proceedings of the Workshop on ITS for Ill-Defined Domains, the 8th Conference on ITS, pp. 1-10, 2006.

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

[12] Ohlsson, S. *Constraint-based student modeling,* in Greer, McCalla, Student Modelling: The Key to Individualized Knowledge-based Instruction, pp. 167-189, Berlin, 1994.

[13] Ormerod, T. C. *Planning and ill-defined problems*, in R. Morris & G. Ward (Eds.): The Cognitive Psychology of Planning, London: Psychology Press, 2006.

[14] Roberts, D.A. *What counts as an explanation for a science teaching event?*, Teaching Edu.,3, pp.69-87, 1991.

[15] Simon, H. *The structure of ill-structured problems*, Artificial Intelligence, No. 4, pp. 181-201, 1973.

[16] Spiro, R. J. et al. *Cognitive Flexibility, Constructivism and Hypertext. Random Access Instruction for Advanced Knowledge Acquisition in Ill-Structured Domains*, in Educational Technology Vol. 31, No. 5, pp. 24-33, 1991.

[17] Taylor, J. & Boulay, B.D. *Studying novice programmers: why they might find learning Prolog hard,* in Rutkowska & Crook (Eds), Computers, Cognition & Development: Issues for Psychology & Education. Wiley, NY, 1987.

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

[19] Voss, J. F. *Problem solving and reasoning in ill-structured domains*, in C. Antaki (Ed), Analyzing everyday explanation: A casebook of methods, pp. 74-93, London: SAGE Publications, 1988.

[20] Voss, J. F. & Post, T. A. *On the solving of ill-structured problems*, in Chi, Glaser, & Farr (Eds), The nature of expertise, Lawrence Erlbaum, 1988.

[21] Weber, G. *Episodic learner modelling,* Cognitive Science, Vol. 20, pp. 195-236, 1996.

[22] White, B. Y. & Frederksen, J. R. *Inquiry, modeling, and metacognition: Making science accessible to all students*, Cognition and Instruction, Vol. 16, No. 1, pp-3-18, 1998.

[23] Xu, S. & Chee, Y.S *Transformation-based diagnosis of student programs for programming tutoring systems,* IEEE Trans on Software Eng, 29, 4, pp. 360-384, 2003.

Appendix A

Exercise 1: A salary database is implemented as a list whose odd elements represent names and even elements represent salary in €. For example: [‘A’, 3600, ‘B’, 5400, ‘C’, 6300,..., ‘D’, 4200]. Please, define a predicate which creates a new salary list according to following rules: 1) Salary less equal 5000€ will be raised 3%; 2) Salary over 5000 € will be raised 2%. The new salary list would be: [‘A’, 3708, ‘B’, 5508, ‘C’, 6426, ..., ‘D’, 4226]. Notice: the representation of 3% and 2% corresponds to 0.03 and 0.02 in Prolog, respectively.

Exercise 2: The income of a company is implemented as a collection of facts in Prolog: invoice(InvoiceNr, ClientNr, Amount, Date) where Amount is a sum of money in old German Mark. Please, define a predicate ***invoice_e/4***, which converts invoices from German Mark into Euro.

Exercise 3: A list represents numbers of audience for a series of TV programs. Each list element contains a sublist with information about the TV station, the program title and the number of audience (in Tsd). The list is ordered in descending order according to the number of audience and is implemented as an argument of the predicate ***audience/1*** in the database of the Prolog system: audience([[TV1, Pro1, 5300], [TV2, Pro2, 4200],...,[TVn, ProN, 3000]]). Please, define a predicate which builds a new list of programs for a given name of TV station. Please notice, that the original order should be kept and the operator ***not/1*** is provided to negate an unification.

Appendix B

Student solution SP1 for Exercise 2:

```
plus([], L2, L2).

plus(Gehalt,L2,R):-
        Gehalt=[Kopf|Rest],
        Rest=[Kopf1|Rest1],
        Kopf1<=5000,
        NKopf1 is Kopf1*1.03,
        plus(Rest1,[L2|Kopf,NKopf1],R);
        Gehalt=[Kopf|Rest],
        Rest=[Kopf1|Rest1],
        Kopf1>5000,
        NKopf1 is Kopf1*1.02,
        plus(Rest1,[L2|Kopf,NKopf],R).
```

Student solution SP 2 for Exercise 2:

```
gehalttarif(Gehaltvorher,Gehaltnachher):-
        gtacc(Gehaltvorher,[],Gehaltnachher).

gtacc([GLvorName,GLvorDM|GLvorTail],Acc,GLnach):-
        gtacc(GLvorTail,[GLvorName,GLneuDM|
Acc],GLnach),
        (GLneuDM is GlvorDM*103/100,Gehalt<=5000);
        (GLneuDM is GLvorDM*105/100, Gehalt>5000).
Gtacc([], Acc, Acc).
```

Student Solution SP 3 for the problem *Compound interest*:

```
invest(S,_,0,S).

invest(S,R,X,End_S):-
        X is New_P-1,
        invest(S,R,New_P,New_S),
        End_S is New_S+R*New_S.
```