Diagnosing Errors in Logic Programming -

The case of an ill-defined domain

Nguyen-Thinh Le

Natural Language Systems Division

Department of Informatics

University of Hamburg

le@informatik.uni-hamburg.de

August 08, 2007

Abstract

Intelligent Tutoring Systems (ITS) 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 constraint-based modelling techniques to describe the solution space of ill-defined problems in logic programming and to diagnose errors in solutions for those problems. This technology has been integrated into a web-based ITS (INCOM) and has been evaluated with student solutions from past examinations.

Zusammenfassung

Intelligente Lernsysteme (ITS) in den letzten Jahren haben große Fortschritte gemacht. Viele von diesen Erfolge wurden hauptsächlich für wohl-definierte Probleme erzielt. Jedoch ist das Lösen von schlecht-definierten Probleme wichtig, weil es die kognitiven, meta-kognitiven und Argumentationsfähigkeiten der Studenten fördert. In diesem Artikel demonstrieren wir, wie constraint-basierte Modellierungstechniken angewendet werden, um den Lösungsraum von schlecht-definierten Problemen zu beschreiben und um Fehler in Lösungen für solche Probleme zu diagnostizieren. Diese Technologie wurde in ein web-basierten ITS (INCOM) integriert und anhand von Studentenlösungen aus früheren Klausuren evaluiert.

Keywords: ill-defined problems, error diagnosis, logic programming, constraint-based modelling, Intelligent Tutoring Systems.

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 [11]. 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 [6, 17]. In solving ill-defined problems, students apply their domain knowledge in a meaningful way instead of storing a chunk of concepts in a memory [25].

2. Enhancement of meta-cognitive skills: ill-defined problems require solvers to control and regulate the selection and execution of a solution process [6, 3, 5]. In the process of solving ill-defined problems, students employ their meta-cognitive skills, such as changing strategies, modifying plans and reevaluating goals in order to reach an optimal solution [25].

3. Enhancement of argumentation skills: since ill-defined problems require students to consider alternative solutions, successful students can provide evidence for their solution [22, 23]. 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 non-determinism of program execution, the underspecification of programming constructs and the concept of recursive programming which is its most important programming technique [20]. 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 [21, 26], program verification [13], plan and bug library [24], model tracing [1] and constraint-based modelling (CBM) [15]. Among these, model tracing is used by cognitive tutors which are among the most successful ITS today [8]. 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 many. 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 the CBM technique and how we apply it to model the solution space for a Prolog problem. The actual error diagnosis is described in the fifth section. The sixth section illustrates the architecture of INCOM briefly. In the seventh section we show our evaluation result. The contributions of our work are discussed in the eighth section. Our conclusions and future works are summarized in the last section.

2. ILL-DEFINED[1] PROBLEMS

There is no formal definition in the literature of what constitutes a "well-defined problem". Instead, we must be 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 exists a limited number of transformation steps which can be relatively easily formalized; 3) evaluation functions are specified and 4) the goal state is unambiguous [7]. If one or several of these conditions are violated, the problem is considered ill-defined [16]. Most researchers agree that a design problem is a representative of ill-defined problems [4], because 1) the start state is underspecified, 2) there is no predefined set of rules for completing the task, and 3) 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" [18]. Therefore, 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, may have well-specified start and goal states, but underspecified transformation rules and 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 [19] and 3) there are only aesthetic value judgments, but no absolute or 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) which should be solved in free-form using our system INCOM.

3. SOLUTION SPACE FOR A LOGIC PROGRAMMING PROBLEM

Before we attempt to define the solution space for a logic programming problem, we first describe the general conditions.

First, in our current prototype the interaction with students is divided into two phases corresponding to the methodology of software engineering:

1. The exercise analysis,
2. The design and implementation.

In the first phase, the ability of problem analysis of the student is determined. At the same time, the student is supported to understand the exercise and to model an adequate signature for the predicate to be implemented. The implementation is carried out in the second phase and its result is subjected to the diagnosis. In addition, the system gains during the diagnosis procedure following information:

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 latter one.

- The implemented strategy of the student solution (i.e. naïve recursion, recursion with decreasing accumulation, …)
- The correct implementation of the required programming techniques (i.e. implicit or explicit unification)
- The correct consideration of general semantic principles of the programming language (i.e. the instantiation conditions for arithmetic evaluations)

As a result of the first phase of the two-tiered interaction design, a specified and verified predicate schema is available as an intermediate result. It contains information about the arity of the predicate, the meaning and the instantiation mode of every argument position. This kind of information not only simplifies the localisation of students' errors in the subsequent implementation step, but also admits the generation of feedback which is related to the intention of the student.

Second, 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. With those general conditions, the solution space of a programming problem is characterized by the following factors:

3.1 Variety of solution strategies

To solve a logic programming problem, there are many alternative solution strategies, for each of them a wide spectrum of solutions can be created. For example, to solve a problem of processing all elements of a list, one can choose between recursion by passing many elements or only one element, in case passing one element, there are several possible alternatives: naive recursion, inverse recursion, recursion using an accumulation predicate or applying the railway-shunt technique.

3.2 Programming techniques and primitives

To implement a solution strategy, we can apply different programming techniques in accordance with the available primitives of Prolog. 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 these 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. Here, we just take the neutral elements: X^1=X and X^0=1 into account. 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 =. A unification subgoal is referred to as an explicit unification. A unification can also occur if two different argument positions share 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 an appropriate relation is defined as a collection of facts in the database. 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)

3.3 General semantic principles

In general, the sequence in which programming constructs are arranged is arbitrary. However, the free arrangement of programming constructs is restricted by general semantic principles. In Prolog they assure that a Prolog predicate definition is executable. The following is a subset of general principles of Prolog, which is not an exclusive list:

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

3.4 The free 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 arbitrary names for variables and predicates as they are used to do without using 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.

3.5 Helper predicates

Helper predicates can be defined according to individual needs. There are two reasons to define a helper predicate: 1) to execute a necessary subtask and enable the re-usability and 2) to keep the code in the main predicate definition simple. Thus, the space of helper predicates which might be defined by each individual is open-ended.

4. Knowledge Representation

4.1 Modelling a Solution Space: the Constraint-based Approach

The CBM approach has been proposed in [15] 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. A constraint is used to describe a fact, a principle or a condition which must hold for every solution contributed by the student. I.e., the following principle can be formulated as a constraint:

Example 1: a Prolog principle

An arithmetic subgoal 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 at the right hand side of an arithmetic subgoal,

Satisfaction: X ought to be bound.

In addition, constraints 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 represents the requirements of the given task. Additional requirements, which have to be satisfied in that specific situation, can be specified in the satisfaction part. 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. The following Constraint 2, for example, examines whether a clause is a recursive clause according to a requirement.

Example 2: a specific requirement

A clause has to be a recursive case.

Constraint 2:

Relevance: Y is a clause with a clause head Functor(Arguments)

Satisfaction: Y has a subgoal in the clause body, which has the same functor and argument list as in the clause head.

In order to evaluate constraints, we define a formal representation for constraints: *constraint(Id, Type, Relevance, Satisfaction, Severity, Position, Hint) where:*

- *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 the relevance part;
- *Satisfaction* is the satisfaction part;
- *Severity* indicates the severity of the constraint, it ranges between zero if the constraint is very important and one if it is just informative;
- *Position* indicates the error location;
- *Hint* is an instructional message, which explains the error.

Syntactically, the relevance part and the satisfaction part are logical expressions, i.e. conjunctions of propositions about a problem state [12]. 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 at three levels: clause level, goal level and argument/functor level. From the horizontal perspective, 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 perspective, following relationships can be detected: 1) the connection between an argument and a goal, 2) the connection between a goal and a clause. 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 obtained, e.g. the instantiation state of an argument or the type of an argument, if a structural element is analysed in the global context of the corresponding program. The instantiation state of an argument is determined using the predicate declaration and the inferred instantiation states for all arguments from the left context of the argument being considered. Such information cannot be read off if only a partial structure of a Prolog program is dealt with.

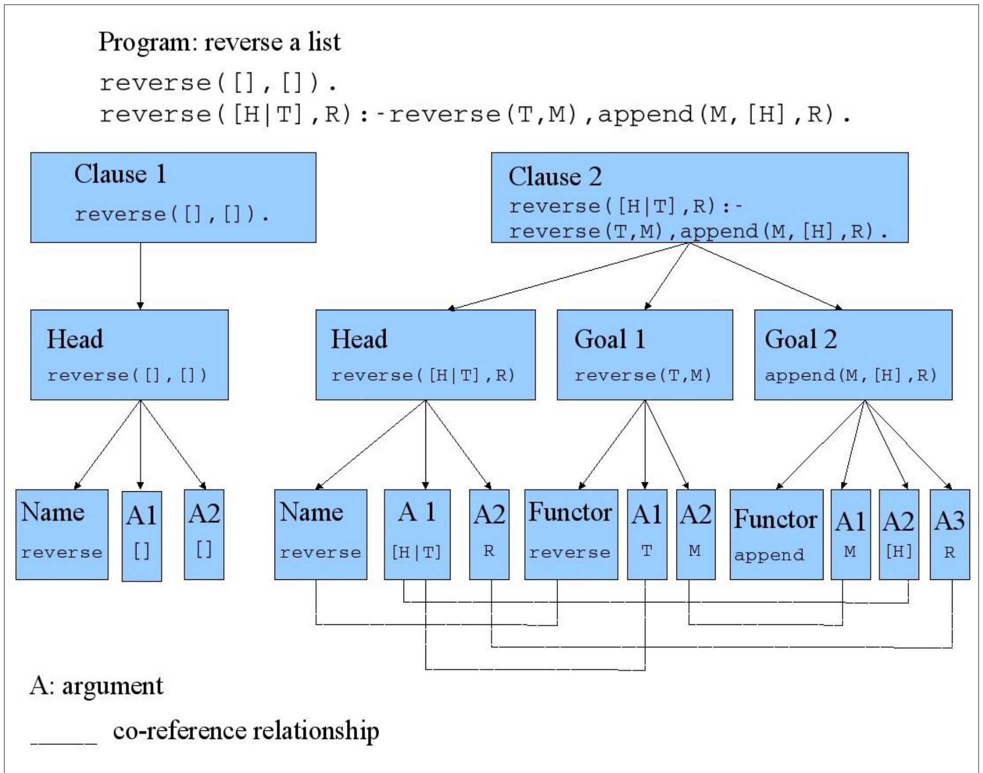

Figure 1: Structure of a Prolog predicate definition

As a result, we need to extract information about horizontal and vertical relationships between structural elements from the student program and its declaration in order to create three types of assertions which later will be used by the relevance and satisfaction parts of constraints:

headarg(Creator, ClauseIndex, ClauseType, AuxiliaryList, HeadName, HeadLength, ArgIndex, ArgType, ArgValue)[2]

bodyarg(Creator, ClauseIndex, SubgoalIndex, Functor, SubgoalType, SubgoalLength, ArgIndex, ArgType, ArgValue)

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

2 Creator: the assertion being considered is created from a student solution or the semantic table (see section 4.3); ClauseIndex: the index of the clause within the program; ClauseType: the clause being considered is a base case or a recursive case; AuxiliaryList: 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; ArgIndex: the index of the argument within the clause head; ArgType: an argument can have one of the types: variable, anonymous variable, list, atom, number, arithmetic or arbitrary; ArgValue: 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, unify, term test, user defined, relation, recursion or unknown; SubgoalLength: the number of arguments the subgoal consists of; InstantiationState: an argument is bound or free.

headarg and *bodyarg* contain information about each argument in the clause head and in the clause body, respectively. If an assertion of the type *argmode* exists, it denotes that the argument is bound, after its corresponding subgoal has been executed [9]. 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: the position of errors, the constraint severity and the hint encoded in the constraint [10].

4.2 Declaration constraints

The declaration constraints examine the arity of a predicate, the type and instantiation mode of each argument position. These constraints are employed exclusively in the first interaction phase in order to diagnose the predicate signature.

4.3 Generalized Sample Solutions and semantic constraints

The generalized sample solutions describe the framework of a predicate definition in relational form in which clauses, subgoals and argument positions are not restricted to a particular sequential arrangement. In addition, all unification conditions are expressed explicitly and clause heads as well subgoals are represented in normal forms (Table 2 and Table 3). The normal form representation reveals the underlying programming techniques and thus, the diagnosis becomes more accurate. A generalized sample solution is specified for each solution strategy, all of which compose a so-called semantic table for a particular problem exercise.

Table 2: A semantic table for a predicate declaration

Argument name	Modus	Type	Synonym names	Description
Arg1	+	list	‘old Salary’,’old’	This argument represents the old salary
Arg2	-	list	‘new salary’, ‘new’	This argument represents the new salary

Table 3: An example of a semantic table for a predicate definition

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

Semantic constraints are constructed based on the semantic table which contains information required to solve a given problem. From the tables above, *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 contain 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 the corresponding student assertions in the selected mapping. 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)*

where s is a subset of semantic assertions S which describe a semantic unit, for instance: an explicit unification; *SP* is a subset of student assertions and *test(SP, variant(s))* tests whether the student solution satisfies any variant of the selected semantic unit *s*. Semantic constraints and semantic tables are used to model the space of solutions which satisfy semantic requirements.

4.4 Pattern Constraints

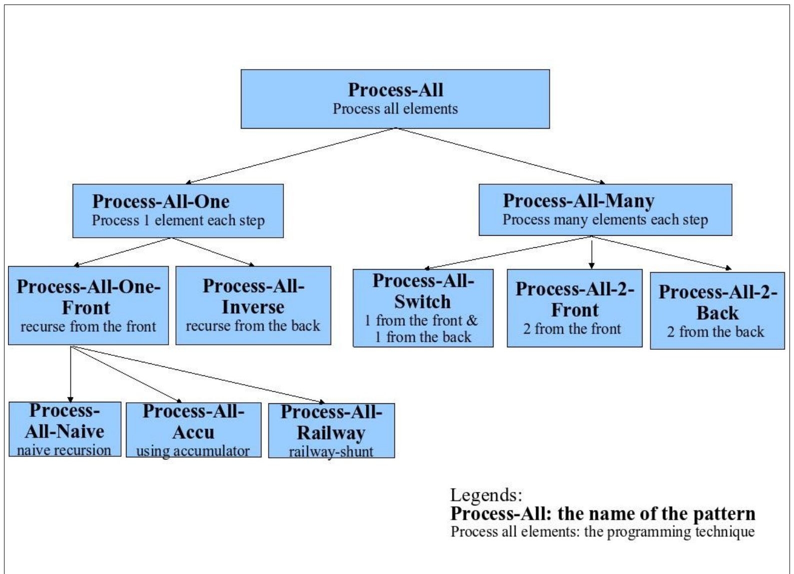

Figure 2: A small hiearchy of Prolog patterns

Standard solution strategies can be modelled as patterns (i.e. Process-All-Elements, Test-All-Elements, ... [8]).

A pattern describes the coarse structure of a possible student solution. Patterns are used to create hypotheses about the strategy implemented in the student solution. For that purpose, the student solution is decomposed into components which are then compared with the pattern corresponding to the exercise type. A pattern encapsulates the application of several programming techniques, which construct the desired semantics of the pattern, i.e. the recursive processing of list elements, the accumulation of partial results. Many programming techniques can be combined and employed typically in different patterns. Therefore, they can be organized in a hierarchy as Figure 2 illustrates. Sub-patterns can inherit the attributes of their super-pattern. Internally, a pattern is modelled by so-called pattern constraints which describe the pattern's attributes and are independent from any problem task. Pattern constraints are partly redundant to semantic constraints. If a pattern constraint is violated, a strategy related remedial hint can be derived and returned to the student. Hence, such constraints can enhance the explanation quality of the diagnostic results but they are not mandatory. Not always a suitable pattern can be found for all possible exercise types and solution strategies. In such a case, no strategy related remedial hints can be given.

4.5 Transformation Rules

A programming technique or a construct can be varied in different ways. Especially, arithmetic expressions can be transformed using the commutative or distributive law. In order to represent a space of alternatives for a programming technique or a construct, we have to define transformation rules.

Currently, we do not consider recursively embedded arithmetic expressions. Our system just copes with arithmetic expressions without nesting, for example: A*(B+C). For such arithmetic expressions, the following transformation rules are required:

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

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

Transformation rules are also required to cover solutions, for which helper predicates are required. If the student wants to keep the code in the main predicate definition clear and simple, then he can define a helper predicate. Applying the unfolding/folding techniques proposed in (Tamaki & Sato, 1984) we are in a position to transform a student solution with a helper predicate to an equivalent one without, if not both the main predicate and the auxiliary predicate apply the recursion technique. The solution without helper predicates is supposed to correspond to one of the generalized sample solutions in the semantic table. If both the main predicate and the helper predicate require recursion as part of the solution strategy, the definition of the helper predicate must be available in the semantic table.

4.6 General Constraints

General constraints express the general semantic principles of the programming language. They are not specific to any problem task and must be adhered by every correct solution. For example, Constraint 1 in section 4.1 represents a general Prolog principle which requires that every variable of the right hand side of an arithmetic subgoal must be bound.

5. Error Diagnosis

The diagnosis is carried out as an interaction of hypothesis generation and hypothesis evaluation. Hypotheses are interpretation variants for the student solution, which are generated by mapping the elements of the student solution to the ones of the generalized sample solution. At the same time, the systematic variations of the sample solution, which are created by applying transformation rules, are also considered. Every interpretation hypothesis is evaluated based on the relevant constraints. In addition, the score of the constraints, which are violated by the selected hypothesis, is computed based on a multiplicative model. That score is used to take a decision for the most plausible interpretation.

Diagnostic information about the shortcomings of the student solution can be gained from constraint violations as well as from the hypothesized mapping between the student solution and the generalized sample solution. In this manner, superfluous and missing elements in the student solution can also be detected.

From the vertical view, the interaction of hypothesis generation and hypothesis evaluation takes place on different levels subsequently:

1. Selection from the alternative sample solutions in the semantic table;
2. Mapping of the clauses;
3. Mapping of the subgoals within a clause;
4. Mapping of arguments and operators;
5. Mapping of summands in a arithmetic expression;
6. Mapping of factors and algebraic signs in a summand.

Suppose, `X` is a set of expressions on the level to be considered, which are extracted from the semantic table. I.e. on the clause head/subgoal level `X` is a set of clause heads and subgoals. `Y` is a set of expressions of the same level which are extracted from the student solution. From the horizontal view, on each level the diagnosis procedure works as follows:

- Create a set `Z` of mappings between `X` and `Y` (see the 1st algorithm).
- If `z∈Z, map(x,y)∈z` and `x` contains transformable expressions, then generate variants for `x` and hypothesize the best variant of `x` (see the 2nd algorithm); Otherwise, go to the next step.
- For each , evaluate constraints based on the mapping `z` (see the 3rd algorithm).
- The best mapping `z` is the one which has maximal plausibility.

1. **Create a set of mappings between `X` and `Y`**

   A mapping is a combination of expressions in `X` and `Y`, where the following cases are considered:

   - If `X` is empty and `Y` is not empty, then take a `y∈Y` and add `map(NIL,y)` to the current mapping.
   - If `Y` is empty and `X` is not empty, then take a `x∈X` and add `map(x,NIL)` to the current mapping.
   - Otherwise, for all `x∈X` and `y∈Y` add `map(x,y)` to the current mapping.

2. **Hypothesize the best variant for the expression `x` in `map(x,y)`**

   - If `x` is a product term, then mathematical transformations are applied to `x` and a set `V` of variants of x is generated.

- For each v∈V, two steps are required: 1) decompose v and y in factors/algebraic signs and 2) evaluate constraints for the mapping m between v and y (see the 3rd algorithm).
- The variant v, which has the highest plausibility, is the best one for expression x.

3. **Evaluate constraints based on the mapping z**

    Select constraints of the current diagnosis level.

    The plausibility for the mapping z is computed by the formulae: `P(z)=S1*S2*…*Sn` where `S1, S2...` and `Sn` are the severity scores of the constraints which are violated and $0<Si<1$.

6. 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.

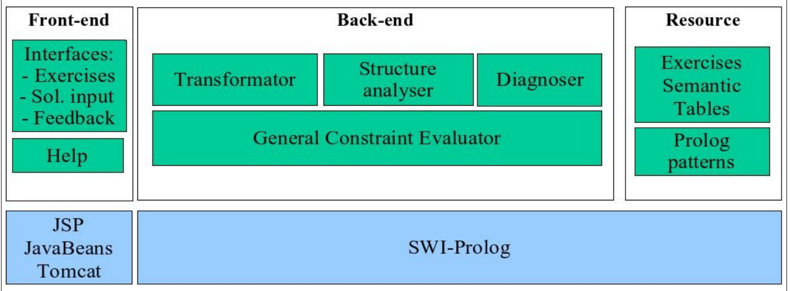

Figure 3: The architecture of INCOM

7. 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 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 was to find out whether the solution space modelled by CBM covers possible student solutions, and thus the ITS was able to give appropriate diagnostic information.

So far 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. highly fragmentary clauses. Furthermore, 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. Test cases are defined based on student solutions.

The following example demonstrates how we added declaration information to a student solution for Exercise 1 in order to define a test case:

A student solution SP1:

```
gehaelter([],[]).
gehaelter([K|Xs],[K|Ys]):-odd(K), gehaelter(Xs, Ys).
gehaelter([K|Xs],[Kb|Ys]):-even(K), K is K*1.03, gehaelter(Xs, Ys).
```

A predicate declaration:

[(sp, gehaelter, [(A1, +, list, New salary list), (A2, -, list, Old salary list)])]

The predicate declaration above has been "guessed" manually and means that the student predicate has a functor name "gehaelter" and two argument positions. The first one represents a "New salary list" and is of type list and of input mode. The second one represents an "Old salary list" and is of type list and of output mode. The declaration information is used to hypothesize the intention of the student, and to understand the subsequent implementation in the student solution.

Table 4: Evaluation of student solutions

Exercise	Year	Participants	Major	Collected solutions	Solutions not covered by INCOM
1	1999	20	Informatics	11	0
2	1999	20	Informatics	5	2
3	2000	39	Informatics & Business Informatics	10	0

We summarize the number of participants who had to solve the exercises (Appendix A) and the number of collected solutions which could be diagnosed by INCOM in Table 5. All exercises are taken from an examination of the second chance[3]. The participant population is students who have chosen their major in Informatics or Business Informatics. The fifth column of the table shows the number of solutions which were sufficiently

3 Students have two chances: the first examination takes place directly after the end of the semester, the second one is normally available one month later. Students, who have not passed or do not attend the first examination, are allowed to participate in the second one.

elaborated for applying a diagnosis. The system INCOM could diagnose almost all collected solutions correctly except two of them (Appendix B, Table 6, Nr. 1 and Nr. 4). In one case the solution contained a disjunction operator *';'* which is currently not supported by the system. Therefore, it returns a message like *"Your solution contains a syntax error"*. In the other case the semantic table was incomplete, since it did not contain a generalized sample solution implementing an accumulator. In this case, the system returns misleading diagnostic information because it interprets the student solution based on a generalized sample solution which is most similar to the student solution.

8. Contributions

In comparison with other application domains, for which constraint-based ITS have been developed, the problem tasks in logic programming demonstrate a relatively challenging task for system development. From the view of a domain model, we are confronted with totally novel requirements, because simplified assumptions, which have been made for other application contexts, are no longer valid. Those assumptions are:

1. the absence of a unique ideal solution,
2. the variance, which arises from the free choice of identifier names,
3. the variance, which results from different solution strategies and implementation techniques,
4. the ability to determine the solution strategy implemented in the student solution,
5. the possibility to divide complex program code into several separate predicate definitions.

According to these five criteria, the other constraint-based ITS published so far provide tasks, which are usually simplified until they become well-defined. Typically, an ideal solution is assumed and constraints serve primarily to enumerate the variations of the solution. This practice fails, if there are many or even uncountably many solutions for a problem task like in the domain of programming. Hence, programming problems should rather be assigned to the class of ill-defined domains, even though any software solution has a formal semantics.

For this situation, we have developed an approach which avoids ideal solutions as far as possible. For the purpose of diagnosis, concrete sample implementations are generalized to abstract schemata, which are then restricted by additional co-reference and general linearisation conditions. Therefore, the advantages of the constraint-based modelling are most convincing in correspondence with the treatment of possible serialisation variants.

In addition, generalized sample solutions in the semantic table can be used as aesthetic criteria to evaluate student solutions. Whenever there is no mapping between an element in the student solution with any element in the corresponding generalized sample solution, the element in the student solution is considered as superfluous. This contributes to the evaluation of aesthetic features which make the domain of programming ill-defined.

In general, CBM tutoring systems are considered to be unable to identify the implementation strategy of a student solution. If a remedial hint makes a wrong assumption about the student strategy, then the feedback can be misleading [12]. This deficiency is compensated in our prototype by the concept of patterns. From a didactic point of view, patterns fulfil two purposes: they can be used to make assumptions about the strategy implemented in the student solution and from those assumptions strategy related feedback can be derived [9]. From the perspective of knowledge representation, the hierarchical organisation of patterns provides advantages with regard to system development and system extension. This is a feature which has not been proposed for the model tracing approach so far and might even be difficult to realise.

Our system provides a two-tiered interaction design which offers advantages with regard to the diagnosis quality. It makes available the predicate schema, thus providing the diagnosis with essential information about the exercise specific allocation of arguments, and therefore considerably reduces the number of mapping hypotheses that have to be analysed. Furthermore, the two phases of solution development correspond to important stages of the programming process: the signature declaration and the implementation of the required semantics.

Our transformation approach starts with a normalised expression from the semantic table and produces semantic-preserving alternatives, which are compared with expressions from the student solution. Thus, the direction of our transformation is different from the one adopted in comparable works, i.e. [2], where expressions to be examined in the student solution are transformed to a normal form, for which a semantic equivalence with a correct solution has to be proven. However, if a student solution has to be transformed before the proper diagnosis, the localization of the error will become considerably more difficult. This results also in a negative effect on the precision and comprehensibility of the resulting remedial hint. Ultimately, our transformation approach considers a wide spectrum of alternative variants. A comparable diversity of variants can probably be modelled only with a lot of difficulties by a limited number of production rules of a model tracing system.

Our transformation is computationally expensive, because it is based on a generate-and-test approach which creates semantic-preserving variants from the normal forms in the semantic table and assesses them by constraint evaluation. On the other hand, our approach avoids complex constraint conditions which might cause one of the following problems:

1. The relevance part can not be satisfied by an incorrect solution because one or several meaningful conditions can not be fulfilled in the given case. This invalidates the constraint.
2. Even if the relevance part can be evaluated to true, a similar problem can occur in the satisfaction part. This causes an imprecise localization of errors.

Such problems would force developers to foresee all possible combinations of errors in student solutions and thus, require error anticipation in the constraint development. This would not correspond to the notion of the CBM approach.

9. Conclusion and Future Works

The solution space for a problem of logic programming is characterized by many factors: the variety of solution strategies, programming techniques and primitives, general semantic principles, the free choice of names for variables and predicates and the usage of helper predicates. The CBM technique can be applied to describe that solution space. We use constraints for different purposes:

1. Constraints without transformation are particularly suited well to describe an expression 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.
2. Constraints with transformation can be applied to expressions, which may have many variants, i.e. an arithmetic expression that can be modified using the distributive or commutative law. This, however, is possible only as long as the correctness of the transformation can be shown and the space of transformation results is finite.

3. Generalized sample solutions and semantic constraints are required if it is not possible to define a transformation between different solution strategies or a transformation rule can not be verified. Semantic constraints examine the student solution based on semantic information from generalized sample solutions.
4. Pattern constraints are used to describe the structure of standard solution strategies. They serve the purpose to provide strategy-related feedback. They are sometimes redundant because similar conditions are already contained in the semantic constraints.
5. Declaration constraints are used to examine the problem analysis of the student.

Our constraint-based diagnosis approach has been realised and integrated into a web-based ITS. The system has been evaluated for coverage 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 represent a complete set of generalized sample solutions. However, the completeness of the semantic table can not be proven. We will extend INCOM with new exercises and test cases to demonstrate the effectiveness of the CBM approach.

References

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

[2] Gegg-Harrison, T. S. *Exploiting program schemata in a Prolog tutoring system*. Durham, North Carolina, 27708-0129, Department of Computer Science, Duke University, 1993.

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

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

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

[6] Jonassen, D. H. *Instructional design models for well-structured and ill-structured problem-solving learning outcomes*, Educational Technology: Research and Development, 45, 2, pp. 65-94, 1997.

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

[8] 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.

[9] 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.), Proceedings of the Workshop on ITS for Ill-Defined Domains, the 8th Conference on ITS, pages 38 - 46, 2006.

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

[11] 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.

[12] Martin, B. *Intelligent Tutoring Systems. The practical implementation of constraint-based modelling,* Phd thesis, University of Canterburry, 2001.

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

[14] Ohlsson, S. *The interaction between knoledge and practice in the acquisition of cognitive skills*, in S. Chipman (Ed.),

Foundations of knowledge acquisition, 1993.

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

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] Xu, S. & Chee, Y.S *Transformation-based diagnosis of student programs for programming tutoring systems,* IEEE Transactions on Software Engineering, Vol. 29, No. 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 Euro. For example: [meier, 3600, schulze, 5400, mueller, 6300, ..., bauer, 4200]. Please, define a predicate which computes a new salary list based on the given one according to following rules: 1) Salary which is less equal 5000€ will be raised 3%; 2) Salary over 5000 € will be raised 2%. The salary list above should be changed to: [meier, 3708, schulze, 5508, mueller, 6426, ..., bauer, 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 and contains: 1) issued invoices: invoice(Invoice_Number, Client_Number, Amount, Date) where **Amount** is a sum of money in old German Mark. Please, define a predicate **invoice_e/4**, which enables a query of money in Euro.

Exercise 3: There is a list of numbers of audience for a series of TV programs. For each TV program, that list contains a sublist with information about the TV station, title of the program and the number of audience (in Thousand) where the entries of that list are ordered in descending order of the number of audience. This list is implemented as an argument of the predicate **audience/1** in the database of the Prolog system: audience([[ard, goldmelody, 5300], [rtl, bloodthirsty, 4200], [sat1, boulette, 3500], [ottifanten_kanal, greif_denzaster, 3300], ..., [arte, Science, 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 not be changed during the computation and you can use the operator **not/1** to negate an unification.

Appendix B

In following tables, we reproduce the student solutions for the exercises above which occurred in past examinations. The student solutions remain as they were except we change the alignment for subgoals to facilitate reading.

Table 5: Student solutions for Exercise 1

Nr	Student solution
1	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).
2	gehalt([N,G|R], [N,G2|R2]):-G>5000, G2 is G*1,02, gehalt(R,R2). gehalt([N,G|R], [N,G2|R2]):-G<=5000, G2 is G*1,03, gehalt(R,R2).
3	gehaelter([],[]). gehaelter([K|Xs], [K|Ys]):- odd(K), gehaelter(Xs, Ys). gehaelter([K|Xs], [Kb|Ys]):-even(K), K bis K*1,03, gehaelter(Xs, Ys).
4	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).
5	gehalt_neu([], []). gehalt_neu([N,B|R], E):- B>5000, B is B+B*0.02, gehalt_neu(R,E), E is [N,B|E]. gehalt_neu([N,B|R], E):- B is B+B*0.03, gehalt_neu(R,E), E is [N,B|E].

Table 6: Student solutions for Exercise 2

Nr	Student solution
1	rechnung_e(Rechnungsnummer, Kundennummer, Betrage, Datum):-
	rechnung(Rechnungsnummer, Kundennummer, Betrag, Datum),
	Betrage is Betrag/195*100.
2	rechnung_e(Rnr, Knr, B_ineuro, D):-B_ineuro is B_inDM*1.95,
	rechnung(Rnr, Knr, B_inDM, D).
3	rechnung_e(Rechnungsnummer, Kundennummer, EBetrag, Datum):-
	rechnung(Rechngusnummer, Kundennummer, Betrag, Datum),
	EBetrag is Betrag/1.95.
4	rechnung_e(RechNr, KundNr, Euro, Datum):-
	rechnung(RechNr, KundNr, Betrag,Datum),
	Betrag is Euro/1.95.
5	rechnung_e(Rechnungsnummer, Kundennummer, Betrag_e, Datum):-
	rechnung(Rechnungsnummer, Kundennummer, Betrag, Datum),
	Betrag_e is Betrag/1.95.
6	rechnung_e(Betrag, Euro, Rnr, Knr):-Zs is Betrag*195,
	Euro is Zs/100.
7	rechnung_e(Rechungsnummer, Kundennummer, BetragEuro, Datum):-
	rechnung(Rechnungsnummer, Kundennummer, Betrag, Datum),
	BetragEuro is Betrag/1.95
8	rechnung_e(R,K,B,D):-rechnung(R,K,B,D),
	B is B/1.95.
9	rechnung_e(Rechnungsnummer, Kundennummer, Betrag_e, Datum):-
	rechnung(Kundennummer, Kundennummer, Betrag, Datum),
	Betrag_e = Betrag/1.95.
10	rechnung_e(RechnungsNr, KundenNr, Betrag/1.95, Datum):-rechnung(RechnungsNr, KundenNr, Betrag, Datum).
11	rechnung_e(Rnr, KNr, BetragE, Datum):-rechnung(RNr, KNr, BetragDM, Datum),
	BetragE is BetragDM*100/195.

Table 7: Student solutions for Exercise 3

Nr	Student solution
1	sender(_,[],[]).
	sender(GSender, [AktS|Rest], [AktS|GSRest]):-AktS=[GSender,_,_],
	sender(GSender, Rest, GRest).
	sender(GSender, [AktS|Rest], GSRest):-not(AktS=[GSender,_,_]),
	sender(GSender, Rest, GSRest).
2	sender(_, [], []).
	sender(X, [[X,Y,Z]|T], [[X,Y,Z]|H]):-zuschauer([[X,Y,Z]|T]),
	sender(X,T,H).
3	sender(Program, Ergebnis):- einsender(Zuschauer, Program, Ergebnis).
	einsender([], _, []).
	einsender([[Prog, Send, Zahl]|RZuschauer], Programm, [[Prog, Send, Zahl], Rerg]):-
	Send = Program,
	einsender(RZuschauer, Programm, RErg).

	einsender([[Prog, Send, Zahl]|RZuschauer], Programm, RErg):-
	Send/== Programm,
	einsender(RZuschauer, Programm, RErg).
4	zusch_sender([], _, []).
	zusch_sender([[Sender, X,Y]|Rest], Sender, [[Sender, X,Y]|Rest2]):- zusch_sender(Rest, Sender, Rest2).
5	sender([], _, []).
	sender([X|Y], XX, Z):- X=[XX|T],
	sender([Y], XX, Z).
	sender([[XX|T]|Y], XX, [X]).
6	sendung_von_sender([], _,[]).
	sendung_von_sender([[Y, Send|Zahl]|T], Y, [Send, Zahl|R1]):-sendung_von_sender(T, Y,R1).
7	sender([], [], _).
	sender([X|Y], [[A|B]|Z], Sender):- [A|B] = X,
	A=Sender,
	sender(Y,Z,Sender).
8	sender([], _,[]).
	sender(zuschauer([X|Y]), Sender, [A|B]):- [Sender|_]=X,
	X=A,
	sender(Y,Sender,B).
	sender(zuschauer([X|Y], Sender, [B]):-sender(Y,Sender, B).
9	sender([], _,[]).
	sender([H|T], S, [H|X]):- H=[S,_,_],
	sender(T, S,X).
	sender([H|T], S,X):- H/== [S, _,_],
	sender(T, S,X).
10	neu([], N, []).
	neu([[F|RST]|Alt], N, [[F|RST]|Neu]):- N2 is N-1,
	neu(Alt, N2, Neu).
	neu([_|Alt], N, [_|Neu, F]):- neu(Alt, N2, Neu),
	N2 is N-1.