International Journal of Artificial Intelligence in Education 19(2009) 381 - 400
IOS Press

Using Weighted Constraints to Diagnose Errors in Logic Programming - The Case of an Ill-defined Domain

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
http://nats-www.informatik.uni-hamburg.de

Abstract. In this paper, we introduce logic programming as a domain that exhibits some characteristics of being ill-defined. In order to diagnose student errors in such a domain, we need a means to hypothesise the student's intention, that is the strategy underlying her solution. This is achieved by weighting constraints, so that hypotheses about solution strategies, programming patterns and error diagnoses can be ranked and selected. Since diagnostic accuracy becomes an increasingly important issue, we present an evaluation methodology that measures diagnostic accuracy in terms of (1) the ability to identify the actual solution strategy, and (2) the reliability of error diagnoses. The evaluation results confirm that the system is able to analyse a major share of real student solutions, providing highly informative and precise feedback.

Keywords. intelligent tutoring systems, error diagnosis, logic programming, weighted constraint-based modeling

INTRODUCTION

Problem solving in ill-defined domains is a human skill of high relevance for many practical settings and therefore deserves due attention in the learning process. There are, arguably, clear model cases of ill-defined domains, like law, architecture or music, where interpretation, aesthetics and opinion play a major role (Lynch, Ashley, Aleven, & Pinkwart, 2006). For others, the distinction between well-defined and ill-defined is much less clear, but we assume that by considering such borderline cases much can be learned about the available options to deal with the ill-defined nature of a domain within a tutoring system. Computer programming is such a borderline case which we are going to study in more detail using logic programming as an example.

After discussing at which level of complexity different criteria for ill-defined domains are applicable, we will present a prototype system INCOM which aims at coaching learners by analysing and commenting on their possibly incomplete solution attempts, providing correction proposals if requested. The system has been designed with advanced beginners in mind, and is geared at helping them to overcome notorious learning barriers. Such blocks are typical for beginners in logic programming, who have already acquired the basic concepts, but fail to apply them in meaningful ways, because they are not yet familiar with the typical solution strategies and idioms of the paradigm and therefore lack viable solution ideas.

In this context, a tutoring system has to deal with all the aspects of ill-definedness. For a learner of logic programming, the most relevant aspects, which might contribute to the ill-defindness, are namely (1) the existence of several alternative solution strategies, (2) the numerous possibilities for semantically equivalent syntactic reformulations, and (3) the use of helper predicates as a means of abstraction and modularization. The system makes use of weighted constraints, where the constraints provide the basic mechanism for error diagnosis, while the weights allow us to choose from among a large set of competing interpretations the one which seems to be most plausible. Finally, we report on a preliminary evaluation of the INCOM system on authentic examination results which shows that the approach is both reliable in identifying the solution strategy and its major constituents, as well as accurate in diagnosing errors found in them.

PROGRAMMING AS AN ILL-DEFINED DOMAIN

The question whether at all, and if so in which respect, programming should be considered ill-defined, obviously depends on the level of task complexity that is aimed at by the learning process. We broadly distinguish five complexity levels with an increasing degree of difficulty:

1. Auxiliary programming concepts that can be trained such as by means of completion tasks or slot filling exercises, usually with a unique outcome:
   - What's the result of the unification `[X,b] = [a|Y]`?
     `X = ..., Y = ...`
   - Add the missing variables in the following predicate declaration:
     `prefix(Prefix,List) :- append(...,...,...)`
2. Highly pre-specified programming tasks where the problem description already narrows down the space of possible solutions to a single one and the system mainly needs to take care of syntactic variants, such as a predicate for finding the maximum of two numbers:

   ```
   maximum(X,Y,X) :- X >= Y.
   maximum(X,Y,Y) :- X < Y.
   ```

   Template-based approaches to input analysis are no longer sufficient for such problems. In order to not reveal to the student too many details of the solution space at a too early point in time, the input mode should be completely unrestricted, thus mirroring a real software development situation as closely as possible. The mapping between a (natural language) problem description and a formal specification becomes increasingly difficult, because even for such simple problems there is an (admittedly small) range of possible solution variants which differ with respect to, for instance, type safety or simplicity.
3. Ambitious programming tasks which allow the application of alternative solution strategies like naive recursion, inverse recursion, tail recursion with an accumulator, or the railway-shunt technique (Hong, 2004). Accordingly, problem solving amounts to a series of design decisions in order to develop a consistent solution. It is exactly this property that makes this level attractive from a problem solving point of view: the student is focused no longer on what might be "the correct solution" but invited to develop one according to her own preferences. Appropriate feedback, however,

can only be given to the student if it refers to a sensible hypothesis about the underlying solution strategy most likely used by the student.

4. Complex information processing tasks like the development of a timetable information system. Although the goal of such an exercise seems to be quite obvious, a great number of design decisions have to be made concerning the choice of appropriate data representations, a suitable method for heuristic search, the number and kind of input parameters considered, the presentation of the output, and so on. Moreover, even the range of services the system could offer depends on individual preferences and is therefore open to debate.
5. Innovative application programs like workplace solutions, which have to fit into an existing environment and serve the special needs of certain users. Here, the identification of these needs and the design of appropriate methods to provide a useful support are among the most important issues. Software construction becomes a predominantly social activity between users and developers. Devising an intelligent tutoring system certainly belongs to this type of activity.

From the perspective of ITS development and tutoring programming, level three seems to be the most attractive one. At this level of ambition, programming is less focused on technical foundations, rather starts to develop into a real problem solving activity. As it will turn out, this is also the level, where programming ceases to be a clearly well-defined domain.

To show this, we apply the five criteria suggested by Lynch and colleagues (Lynch, Ashley, Aleven, & Pinkwart, 2006) for identifying ill-defined domains to the different complexity classes above. According to the authors, a domain can be considered be ill-defined if 1) it lacks unambiguous criteria to verify the validity of a solution, 2) the development of formal theories is not compatible with the nature of the domain, 3) even at a novice level, problem solving amounts to a design activity, 4) there are open-textured concepts, which cannot be applied to a concrete situation easily and 5) subproblems, into which a task might be decomposed, are not independent.

The application of these criteria to the domain of programming yields a fairly picture. Having the formal semantics of a programming language available, the validity of a student solution can easily be checked against the requirements of the problem task by means of a suitably chosen set of test cases. This, however, holds only as long as the programming problem is simple enough to be covered by a comprehensive test suite and therefore is not applicable to levels four and five. In this sense, elementary level programming is well-defined. A similar conclusion can be drawn from the second and fourth criterion which concern the availability of a formal theory for the particular domain and the existence of open textured concepts. While the underlying formal theory of programming is exactly the semantics of the machine model at hand and therefore directly available to all problems which address the formal aspects of programming, the existence of open textured concepts is a phenomenon typically associated with programming tasks at level five. Here, terminological clarifications and the development of a formal model for the existing workflow necessities are an integral part of the problem solution proper. A special kind of open-textured concept can also be identified for tasks at a lower level already, if meta-level issues like efficiency, simplicity, elegance, ease-of-use, and so on are considered. In general, however, dealing with such kinds of concepts is clearly beyond the possibilities of current ITS. Instead they usually are tried to be "engineered away" by giving a maximally precise problem statement formulation. Such a simplification, however, is only available for the rather well specified tasks up to level three. And even for them there is no absolute certainty that alternative (mis-)interpretations can be completely ruled out.

Table 1
Arguments in favour of programming being considered ill-defined

	auxiliary techniques	prespecified tasks	ambitious tasks	complex tasks	innovative solutions
no formal validation				**	***
no formal theory				***	***
design activities		*	***	***	***
open textured concepts		(*)	(*)	(**)	***
no independent subproblems			**	***	***
no uniquely specified start state(s)				**	***
no formally specified state transitions		*	***	***	***
no evaluation function for states		*	**	***	***
no uniquely specified goal state(s)			*	**	***

*: marginally relevant, **: relevant, *** highly relevant, (.): relevant only on the meta-level

Evidence for ill-defined aspects in the domain of programming is mainly contributed by the third and fifth criterion, which check for design activities and the independence of subproblems. Certainly, programming is a constructive activity on almost all levels of complexity. While its outcome (the program text) can be subjected to a rigorous formal validation (if a formal specification or test suite exists at all), there is no comparable formal theory available to guide the program development. In contrast to other design activities, such as architecture, only a limited set of discrete "building blocks" can be made use of, though these can be combined in many different ways. As soon as a choice among alternative solution strategies and implementation techniques becomes part of the task to be solved, the specifics of programming as a true design issue can be clearly identified. While making such decisions, expert programmers are guided by (semi-)formal means, like standardized patterns, programming schemata and programming techniques. Beginners, however, tend to be less systematic (Pintrich, Berger, & Stemmer, 1987).

By looking at the internal interdependencies of a problem solution, the fifth criterion, finally, provides additional evidence in favour of considering programming for more ambitious tasks as being ill-defined. Decomposing a given problem into subproblems usually creates very many dependencies between them, such as between different clauses of a predicate, which usually implement complementary aspects of the problem solution, or between the head of a clause and the recursive subgoal in its body. A certain choice in one place normally has major consequences elsewhere in the program. Indeed, given the omnipresence of such dependencies, it is one of the major goals of good program design to isolate different parts of a solution from each other through abstraction and encapsulation.

A quite similar picture results from applying the criteria proposed by Simon (1973) who considers the distinction between well-definedness and ill-definedness not as a property of a particular domain, but takes the perspective of the problem solving process which is imagined as a heuristic search procedure. To qualify as a well-defined problem the solution process must be characterized by the existence of uniquely specified start and end points, as well as a formal transformation procedure for problem states and an evaluation function for them. Again, it is the lack of a formally specified procedure capable of transforming a given initial state into one of the admissible goal states that introduces aspects of ill-definedness into programming tasks of higher complexity.

Besides these aspects related to problem solving itself, there is yet another issue completely ignored in the above discussion: the necessity to present programming assignments as natural language text to the student. Given all the potential pitfalls of language understanding (in terms of deriving a formal problem specification from a textual one) this introduces an additional dimension of ill-definedness into the overall task. For the purpose of ITS design, however, this issue is usually neglected and a common understanding of the task description is assumed. Under this assumption the solution space of a well-specified programming problem is given by the following characteristics:

1. the existence of different solution strategies,
2. the existence of semantically equivalent syntactic reformulations including the choice between alternative implementation techniques (e.g., explicit vs. implicit unification),
3. the existence of alternative sequential orderings of programming constructs,
4. the option to introduce identifiers according to individual preferences, and
5. the possibility of decomposing the problem solution into simpler functional units using either built-in or user-defined helper predicates.

The size of this space, which in general is open, determines the degree to which a given problem needs to be considered ill-defined. In this paper, we consider only the domain of programming tasks of difficulty level 3 and their solution space is determined by the five characteristics above, for example, the **Investment** problem: *A sum of money increases with a constant yearly interest rate. Write a predicate to calculate the sum of money after years of investment.*

STATE OF THE ART

Currently, two approaches that have the potential to be employed in ill-defined domains are model-tracing and constraint-based modelling (CBM). In a model-tracing ITS, error diagnosis and instruction are carried out on the basis of an expert model. This model represents one or more *"ideal"* solutions to a given problem. Whenever a student solution deviates from the expert model, the system provides feedback based on buggy rules. Model-tracing tutoring systems have proven to be successful in well-defined domains such as physics (VanLehn, Lynch, Schulze, Shapiro, Shelby, Taylor, Treacy, Weinstein, & Wintersgill, 2005) and also in the programming domain, LISP for example (Anderson & Reiser, 1985). However, this LISP tutor provides programming tasks of complexity level two because the programming process is guided by templates to be filled in to reach a solution. Thus, the students do not have the possibility of implementing alternative solution strategies. At present, there have been some attempts to apply model-tracing techniques in ill-defined domains: the CATO system (Aleven, 2003; Ashley, 2000), for example, which helps beginning law students acquire basic argumentation skills, and PETE (Goldin, Ashley, & Pinkus, 2001) which tutors engineering ethics.

While model-tracing systems are based on ideal solution models, CBM systems are built from sets of constraints. A constraint represents a domain principle or specifies a task requirement (Ohlsson, 1994). CBM has been applied to tutoring systems in a variety of domains among which the SQL tutor is the most successful one (Mitrovic, Suraweera, Martin, & Weerasinghe, 2004). Typically, the system provides a template-like interface which is very helpful for SQL novices. Although the SQL tutor allows the students to choose between three different solution strategies (using SELECT, JOIN, or nested SELECT in the WHERE clause) (Martin, 2001) and different syntactical reformulations, the students do not have

the possibility of changing the sequential order of SQL constructs or defining subfunctions. Under this condition, the system neglects important learning objectives, such as the proper arrangement of constructs within an SQL statement. The CBM approach has also been applied to a procedural domain, for example database normalisation (Mitrovic, 2002; Mitrovic & Ohlsson, 2006). The tasks, however, are simplified to only deal with fixed-sequence problems (Kodaganallur, Weitz, & Rosenthal, 2006) where aspects of ill-definedness play no significant role. Recently, the CBM approach has been used to develop an ITS in the domain of legal reasoning, a clearly ill-defined domain, in order to coach students in learning argumentation based on a set of solution preferences (Pinkwart, Aleven, Ashley, & Lynch, 2006).

To solve the problem **Investment** using pure Prolog[1], the following strategies can be applied: 1) *Analytic strategy*: the profit of investing a sum of money X with an yearly interest rate of M after Y years is calculated based on a mathematical formulae: End_sum = X*(M+1)^Y; 2) *Tail recursive strategy*: a variable can be used to accumulate the sum of investing money and its interest after each year; 3) *Recursive and arithmetic_before strategy*: the calculation of the profit of investing a sum of money goes back year after year to the first year of investment, then the profit of each year by summing up the investing money and its interest is determined; 4) *Recursive and arithmetic_after strategy*: first, the return is calculated recursively on a new period, then the new period is checked as to whether the old period is an increment of the new one. Following this strategy, a new period is not calculated, but rather tested. For each solution strategy, several semantically equivalent variants might be implemented, depending on the set of available programming constructs of each programming language. Taking only the commutative and distributive laws for multiplication and addition into consideration, the analytic strategy can be implemented in 4 variants, while for each of the three recursive strategies there are 512 variants (Figure 1). Even more variants can be produced by applying other equivalence transformations.

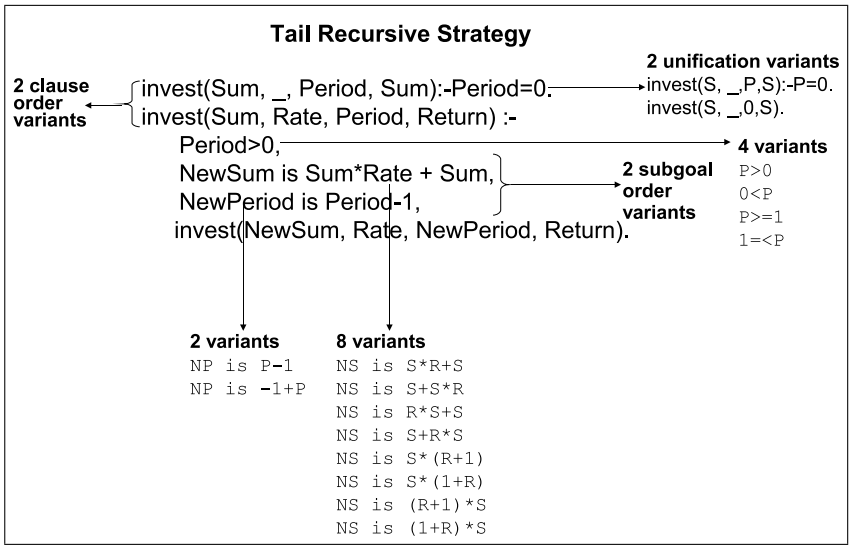

Fig.1. Implementation variants for the tail recursive strategy.

[1]Pure Prolog does not support higher-order built-in predicates

In principle, the model-tracing approach could be applied to model all of the solution paths for the problem **Investment**. However, this would be very laborious because we would have to specify each solution path with appropriate task specific production rules.

The knowledge module of a CBM system does not require one to model all alternative solution paths for a problem (Mitrovic, Koedinger, & Martin, 2003). Instead, it tries to identify the principles of the application domain, in our case logic programming, and the properties of correct solutions for a specific problem. Whenever these properties are available, they can be modelled by means of constraints. A constraint is formalized as an ordered pair consisting of a relevance part and a satisfaction part: $C =< C_r, C_s >$ (Ohlsson, 1994). Here, C_r specifies the relevant circumstances under which the constraint applies, and C_s is a satisfaction condition which has to be met in order for the constraint to be satisfied. If a constraint is violated, it indicates that the student solution does not conform to the principles of a domain or it does not meet the requirements of the given task. Constraints are particularly useful, if they capture a range of possible alternatives, that is if they generalize across a systematic varia-tion of admissible solutions. However, this kind of constraint representation suffers from a number of shortcomings, among them:

1. Severe errors cannot be distinguished from minor ones. If many constraint violations have been obtained for a student solution, there is no means to select among them and to decide on the order in which to present them to the student.
2. In the case of diagnostic ambiguity, no arbitration between alternative error explanations is possible. Diagnostic ambiguity can always be expected if a certain error can be removed in quite different ways. If, for instance, a test of a number being a positive one has been implemented as `X>=0`, the error can be corrected by changing either the operator or the operand. A more serious kind of diagnostic ambiguity is introduced by the existence of alternative solution strategies. Many constraints are valid only in the context of a particular solution strategy. If this dependency is not explicitly modelled, diagnostic information that is derived from a violated constraint might be deceptive to the student (Martin, 2001). Kodaganallur and colleagues attribute this drawback to the complexity of the goal structure that defines the solution process: *"As the problem goal structure complexity increases, it becomes increasingly difficult for a CBMT*[2] *to fathom where in the problem-solving process the student has made an error and in turn it becomes increasingly difficult for a CBMT to offer targeted remediation"* (Kodaganallur, Weitz, & Rosenthal, 2005). To overcome this difficulty, constraints need to be sensitive to different solution strategies and error explanations must be ranked according to some heuristic criterion, such as plausibility.
3. Highly specific constraints, whose relevance part specifies many conditions, become less sensitive to erroneous situations, because their relevance part is not robust against minor deviations. A complex constraint with a conjunction of conditions in the relevance part becomes irrelevant for a student solution if only a single one of the conjuncts fails. Hence, the constraint is satisfied even though this undesired result might have been caused by another error elsewhere in the student solution (Le, 2006). This leads to the paradox that constraints that are meant to diagnose errors can be completely neutralized by other errors and it might even happen that two or more errors mutually neutralize their relevant constraints. In such a case, one would be better off with simpler

[2]CBMT is abbreviation for constraint-based modelled tutor

but *approximate* constraints.

INCOM: A WEB-BASED HOMEWORK ASSISTANCE FOR LOGIC PROGRAMMING

The system INCOM is intended to help students of the course Logic Programming at the University of Hamburg. It is meant to coach students individually solving their homework assignments to better prepare them for classroom activities. The system informs the student about possible errors occurring in her solution attempts and remedial hints are given to improve the solutions.

Two-Stage Coaching

Many studies provide evidence that novice programmers normally tend to start keying in a program right from the beginning without careful consideration of the available programming strategies and techniques (Pintrich, Berger, & Stemmer, 1987). Consequently, they mainly struggle with higher level programming concepts (Pennington, 1987). In addition, one of our own studies (Le & Menzel, 2006) showed that our students lack the ability to properly extract formal requirements from a given problem statement. Therefore, we decided to explicitly support the stage of analysing a problem prior to implementation.

INCOM follows a two-stage coaching strategy: analyse the problem first, then implement a solution. During the initial analysis phase, the student is requested to input an adequate signature for the predicate to be implemented. A predicate signature consists of a predicate name, required argument positions and corresponding argument types as well instantiation modes. If the signature is not yet appropriate, the system provides corrective feedback by highlighting keywords in the task statement and suggests the students exhaust the information given in the problem statement. This analysis stage not only encourages the students to practice analysing problems, but also provides valuable information about the number and the meaning of arguments, which helps to make the subsequent diagnosis in the implementation more accurate.

As soon as a satisfactory predicate signature is available, the second (implementation) phase is entered and the student is invited to compose a solution for the given task using pure Prolog, a version that does not support higher-order predicates, among them cuts, disjunctions, if-then-else operators, assert, retract, abolish and other database-altering predicates. Helper predicates are indicated as part of the problem statement or can be defined by the student herself. There are two cases where a helper predicate is necessary: 1) modularizing a program and 2) defining an accumulative function. INCOM is intended to support helper predicates of the latter category because they are one of the concepts which should be learnt and tutored. In general, the system neither requires that the student adheres to a single solution strategy nor does it prescribe a particular arrangement of solution elements (i.e., no input templates are used).

BRIDGE (Bonar & Cunningham, 1988), a tutoring system for Pascal, also supports a stage-based coaching strategy where the programming process is divided into three phases. In the first phase, the system helps the students to analyse a problem statement and translate it into programming plans using pre-specified natural language phrases. In the second phase, the students construct a visual solution by piecing individual plans together. In the final phase, the students are allowed to translate the programming plan structure into a specific programming language such as Pascal. Our system INCOM differs from BRIDGE by the representation of task analysis in the first stage. Where the task analysis in INCOM

results in a signature for a predicate, users of BRIDGE have a programming plan represented in informal natural language.

Weighted Constraint-Based Error Diagnosis

In the last section we identified some inherent weaknesses of the CBM approach. In order to apply CBM to build a tutoring system for logic programming, a partly ill-defined domain, we enhance this approach with the following components: weighted constraints, a semantic table and transformation rules.

Instead of the *"ideal solution"* concept used in other tutoring systems, the semantic table represents solution strategies for a given problem. Each solution strategy is described by a set of semantic elements required by its implementation. If a semantic element contains an arithmetic expression, transformation rules can be used to generate all possible implementation variants. Weighted constraints establish a mapping between the student solution and the requirements of the semantic table and check general well-formedness conditions.

Together the semantic table, constraints and transformation rules span a fairly large solution space for a programming problem. Weighted constraints can be violated and each violation provides diagnostic information that can be communicated to the student by means of appropriate feedback messages. Constraints are weighted to facilitate a comparison of competing error explanations. This allows the system to hypothesise the solution strategy possibly pursued by the student, and to examine the semantic correctness of her solution. Beyond that, constraint weights can be used to determine the order in which feedback is presented to the student.

MODEL COMPONENTS

Semantic Table

The concept of the semantic table comprises two ideas: 1) it describes several solution strategies, and 2) it represents model solutions in a generalized form, thus covering different implementation variants. For the sample problem **Salary** (see Appendix), the *semantic table* contains two kinds of information: besides a signature for the predicate to be implemented (Table 2), it mainly consists of a set of *generalised* sample solutions (GSS) that describe the semantic requirements of a predicate definition in relational form. That is, clauses, subgoals and argument positions are not restricted to a particular sequential ordering. In addition, all unification conditions are expressed explicitly and clause heads as well as subgoals are represented in normal form (Table 3). The normal form representation reveals the underlying programming techniques. Thereby, the diagnosis becomes more adequate on the conceptual level and resulting feedback more useful. A generalised sample solution is defined for each solution strategy, all of which describe the basic requirements for solving a particular exercise. Table 2 specifies that a predicate to be implemented needs two argument positions, one of them in the input mode and another one in output mode. On both argument positions, a datatype list is expected. Table 3 shows that the declared predicate **p** consists of three clauses. The first one represents a basecase, the other two are recursive cases. The second entry of this table, for example, can be interpreted as follows: the recursive clause requires the existence of a list decomposition for the first argument position OL, a list composition subgoal for the second one, an arithmetic test, a calculation and a recursive subgoal. The

Table 2
A predicate's signature for the task Salary specified in the semantic table.

Name	**Argument**	**Type**	**Mode**	**Synonyms**	**Description**
p	A1	list	input	old list, salary database	This argument stands for the old salary list
p	A2	list	output	new list, new salary database	This argument stands for the new salary list

Table 3
Semantic requirements for implementation of the task Salary

Clause	**Head**	**Subgoal**	**Description**
1	p(OL,NL)	OL=[]	Basecase list empty
		NL=[]	New list also empty
2	p(OL,NL)	OL=[N,S|T]	N:Name; S:Salary
		NL=[N,Sn|Tn]	Build a new list
		S=<5000	Salary =< 5000
		Sn is S+S*0.03	Salary is increased
		p(T, Tn)	Recurse old list
3	p(OL,NL)	OL=[N,S|T]	N:Name; S:Salary
		NL=[N,Sn|Tn]	Build a new list
		S>5000	Salary>5000
		Sn is S+S*0.02	Salary is increased
		p(T,Tn)	Recurse old list

dependencies between the arguments are represented implicitly by the coreference requirements between these subgoals.

Weighted Constraints

Since for a problem there might be many alternative solution strategies, constraints, which are soly based on a binary logic (violated or not), do not contain sufficient information to select the most plausible solution strategy and the most relevant error hypotheses from a multitude of competing ones. Therefore, we enhance the capability of the CBM approach by attaching a weight value to each constraint. Constraint weights are taken from the interval $[0, 1]$ which can be conceived of as a measure of importance with 0 indicating the most important requirements (Table 4). Given the fact that a clause contributes more information to the overall correctness of the solution than an argument or a functor, a constraint which examines an argument should be specified as being less important compared to a constraint checking a subgoal. We use weighted constraints for the following purposes:

Declaration constraints define requirements for the signature specification, that is the number of

Table 4
Proposed constraint weights

Constraint Weight	Checking Issues
0.01	Clause existence
0.1	Subgoal existence
0.3	Correctness of comparison operators
0.5	Argument existence, position, co-reference
0.7	Subgoal order, factors of a multiplication term
0.8	Correctness of nominator, denominator of a fraction term, anonymous variables

argument positions of the predicate, as well as the type and instantiation mode of each argument position. They are used in the preceding task analysis phase.

General constraints express general semantic principles of the programming language. They are not specific to any task and must be satisfied by any correct logic program. General principles of this kind are, for instance, the instantiation requirements for arithmetic expressions.

Semantic constraints examine whether a solution fulfills the requirements of the task description. Semantic constraints have access to the information from the semantic table. Given a generalised sample solution, the relevance part of a semantic constraint refers to a component of a predicate definition and compares the student solution with respect to that selected component. The set of semantic constraints consists of two groups: 1) *pre-implementation* constraints which check whether an implementation is appropriate according to the information provided by the predicate declaration, and 2) constraints which are classified according to particular constructions of a logic program: *clause head, decomposition, recursion, arithmetic test, calculation, arithmetic term, factor, query, term test* and *unification*. For example, the following semantic constraint is of type *arithmetic test*, has a weight of 0.1 and is defined to examine whether an arithmetic subgoal (e.g. `X<1`) specified in the GSS also exists in the student solution (STS):

IF	An arithmetic test subgoal, which compares a variable X with a number, is specified in the GSS
THEN	A corresponding subgoal comparing a variable SX with a number exists in the STS
WEIGHT	0.1
HINT	In the clause body, an arithmetic test subgoal for variable SX is missing.

In contrast, a constraint checking the internal correctness of an arithmetic expression is weighted as being less important (0.3). It generalizes across a range of arithmetic comparators (`<, >, =<, >=, >, <, >=, =<`) and their commutative variants. For example, both cases $X < 1$ and $1 > X$ should satisfy that constraint:

IF	SX and SY are the operands of an arithmetic test subgoal in the STS, which corresponds to a subgoal $X\ OP\ Y$ of the selected GSS
THEN	In the STS, the operator is identical to OP and SX and SY correspond to X and Y or the operator is reversed and SX and SY correspond to SY and SX
WEIGHT	0.3
HINT	The operator is not correct or the arguments SX and SY have been swapped.

Pattern constraints finally are used to model standard solution strategies. A pattern describes the coarse structure for a class of solutions that are based on the same solution strategy. Patterns are independent of any task. A pattern encapsulates the application of several programming techniques which result in the desired computation, for example, the recursive processing of list elements. Patterns can be organised in a hierarchy (Figure 2) where a sub-pattern can inherit the characteristics of a super-pattern and contain new techniques.

Patterns are used to create hypotheses about the strategy implemented in the student solution and provide strategy-related feedback. Therefore, pattern constraints are partly redundant to semantic constraints. They can enhance the explanatory quality of the diagnostic results but they are not mandatory.

A suitable pattern cannot always be found for all possible exercise types and solution strategies. In such a case, no strategy related feedback can be given (Le, 2006).

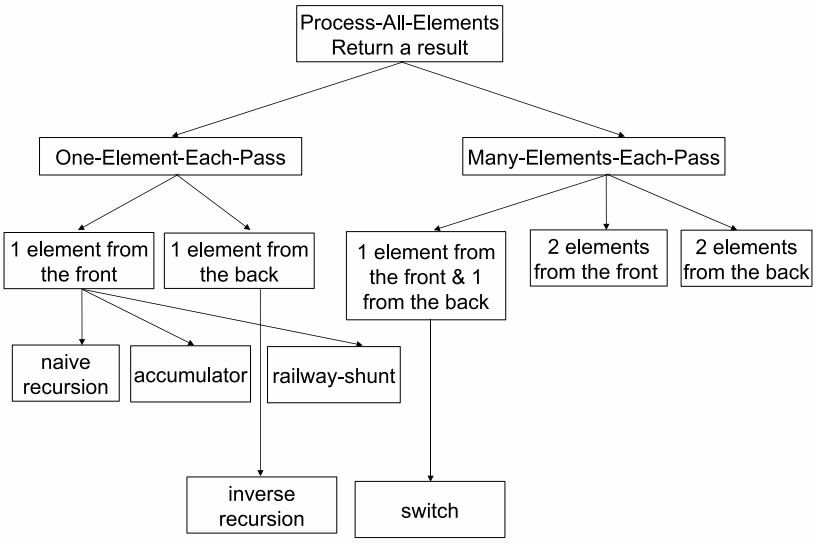

Fig.2. A small hierarchy of Prolog patterns.

Transformation rules

A programming technique or a construct can be instantiated in many different ways. Especially, arithmetic expressions exhibit a great variety of equivalent formulations. In order to represent the resulting space of alternatives in a general manner, we have defined transformation rules, which cover the most important practical cases. Transformation rules produce semantically equivalent distributive and associative reformulations which then can be checked against the student solution. For example, $X * 5 + X * 4$ will be transformed to $X * 9$ or $5 * X + X * 4$, and other variants. Currently, INCOM is not able to handle nested arithmetic expressions.

ERROR DIAGNOSIS

According to the two-phase design of the training scenario, the diagnosis procedure is separated into two steps: signature and implementation diagnosis. Invoking declaration constraints, the signature diagnosis examines the appropriateness of the predicate declaration. Semantic, pattern, and general constraints are activated in the subsequent implementation diagnosis, which determines the implemented solution strategy, the correct implementation of the required programming techniques and the correct application of general principles of Prolog.

In principle, diagnosis is carried out as an interaction of hypothesis generation and hypothesis evaluation. It is exactly this close interaction which allows the system to identify the underlying solution

strategy even in the presence of severe programming errors. Hypotheses are interpretation variants for the student solution. They are generated by iteratively mapping *semantic elements* of the student solution to the ones in the semantic table on all the different levels of the program structure: signatures, solution strategies, clauses, subgoals within a clause, arguments and operators, summands in an arithmetic expression, as well as factors or algebraic signs within a summand. That means, the mapped elements from both the student solution and the semantic table belong to the same semantic class, that is arithmetic subgoal or arithmetic operand.

Every mapping hypothesis **H** is evaluated based on the relevant constraints. The scores of all violated constraints are multiplied into an overall score for the particular mapping: $Plausibility(H) = \prod_{i=1}^{N} W_i$ where W_i is the weight of a violated constraint. That score is used to decide on the most plausible interpretation. For example, if in the context of **H** many severe constraints (i.e., W=0.01) are violated, the plausibility of this hypothesis will be close to zero and less likely to be selected.

Diagnostic information about shortcomings of the student solution can then be gained from constraint violations as well as from the mapping (i.e., superfluous or missing elements). Note that the diagnostic abilities of the system are strictly limited by the completeness of the entries in the semantic table. In case a student follows an unexpected but correct strategy, the system chooses from the available generalized sample solutions the one that is most similar and "forces" diagnosis on it.

According to the sequence of diagnosis stages, feedback of the task analysis is presented to the student first followed by the feedback of the implementation diagnosis. During the implementation phase, feedback is again displayed in two steps: 1) feedback of the type *pre-implementation* which considers the appropriateness of the implementation with respect to the declaration, and 2) feedback derived from general constraints or other semantic constraints (see Figure 3). In each step, feedback is ranked according to the constraint weights.

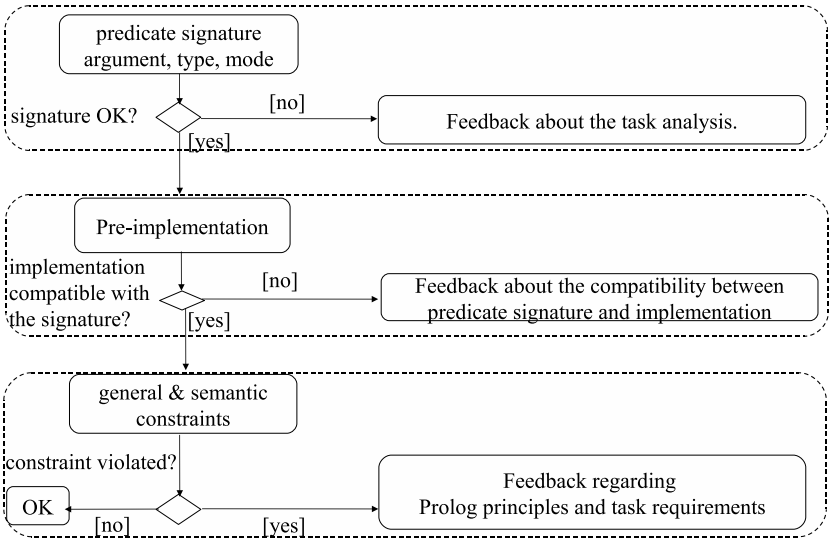

Fig.3. The diagnosis flow of INCOM.

EVALUATION

Diagnostic Accuracy

To validate a particular tutoring system an evaluation is indispensable. Such an evaluation can be carried out in different ways and on different levels, such as internal vs. external evaluations, depending on the development phase the system currently is in. During development, the diagnostic accuracy is of utmost importance because it is the foundation on which a student model is built and feedback produced.

Internal evaluations with respect to diagnostic accuracy are rarely considered when evaluating an ITS. Instead, we can find examples of external evaluation methodologies which are based on comparing the learning effectiveness between a control and an experimental group or on the difference between the results of a pre- and a post-test e.g., (Anderson, Corbett, Koedinger, & Pelletier, 1995; VanLehn, Lynch, Schulze, Shapiro, Shelby, Taylor, Treacy, Weinstein, & Wintersgill, 2005). One of the reasons for this tendency might be that so far mostly well-defined domains have been considered, for which the evaluation of diagnostic accuracy is not an issue. In such a domain, a student solution has a clear structure and thus, elements of a student solution can be mapped directly to the information given in the problem statement. No alternatives need to be considered.

This situation is different in domains where the student is invited to produce a solution within an open space of possibilities. Here we need to distinguish the two aspects of diagnosis: 1) whether the system was able to correctly determine the solution strategy chosen by the student and to identify its components (accuracy of intention analysis) and 2) whether it was able to correctly diagnose the errors in the student solution (diagnostic reliability).

To determine the accuracy of the intention analysis we selected seven exercises and their solutions from past written examinations. Table 5 presents the required skills of those exercises and their complexity in terms of number of clauses and subgoals. The programming tasks are:

1. Access to specific elements within an embedded list.
2. Querying the data base and applying a linear transformation to the result.
3. Modification of all elements of a list subject to a case distinction (see the problem statement in the Appendix).
4. Creation of an n-best list from a data base.
5. Computing the sum of all integer elements of a list.
6. Counting the number of elements in an embedded list.
7. Finding the element of an embedded list that has the maximum value for a certain component.

The participants of examinations were students who had chosen their major in different branches of Informatics. The examination candidates had attended a course in Logic Programming that was offered as a part of the first semester curriculum. It turned out that the system was able to analyse 87.9% (sd=17.1%) of 221 collected student solutions correctly (Le & Menzel, 2008). Note that, since written examinations are fundamentally non-interactive, the scenario from which the student solutions have been drawn differs remarkably from the setting an ITS is devised for. It is not quite clear at the moment what kind of bias was introduced by this approach: on the one hand, a written examination encourages more careful thinking, on the other hand interactive refinement of a program might help to avoid really complex error situations. As a consequence, this evaluation can only be considered a first step and needs to be

Table 5
Degree of difficulty of the examination's exercises

Task	Strategy	Required skills	# clauses (subgoals)
1	1	recusion, list decomposition, term comparison	3 (10)
2	1	calculation	1 (2)
3	1	recursion, list (de)-composition, arithmetic comparison, calculation	3 (12)
4	1	recursion, arithmetic comparison, calculation	3 (9)
5	1	recursion, arithmetic calculation, list decomposition	2 (5)
	2	recursion, arithmetic calculation, list decomposition	3 (5)
6	1	recursion, arithmetic calculation	2 (5)
	2	recursion, arithmetic calculation	3 (5)
7	1	recursion, list (de)-composition, arithmetic comparison	3 (9)
	2	recursion, list (de)-composition, arithmetic comparison	4 (10)

Table 6
Categories for precision, recall

	Should-be bugs	Should-not-be bugs
Retrieved bugs	A	B
Not retrieved bugs	C	

complemented with additional studies involving students interactively working with the system.

To measure diagnostic reliability we can apply well-known measures from information retrieval (Rijsbergen, 1979), since both are subset selection problems sharing the same kind of error characteristics: overgeneration (too many bugs have been reported) or undergeneration (too few bugs have been reported). In such a situation, recall and precision are appropriate quality measures, which are defined with respect to Table 6 as follows:

- $Recall = \frac{A}{A+C}$
- $Precision = \frac{A}{A+B}$

Under these definitions, a high precision means that the model is based on fairly reliable constraints which have a low risk of producing false alarms, that is, the developer was careful to avoid particularly risky constraints. A high recall, on the other hand, means that the diagnosis has a good coverage, that is, it considers a sufficiently rich set of relevant constraints.

Since a constraint can be relevant and applied to different structural elements of a solution, several bugs might stem from the same constraint. Whereas the categories *retrieved* and *not-retrieved* bugs are produced by the system's diagnosis component, *should-be* and *should-not-be* bugs need to be determined by human judges. Therefore, we need to define a gold-standard against which the diagnostic reliability can be measured.

Unfortunately, it is not as easy to develop such a gold-standard. Adopting individual perspectives or preferences, experts tend to disagree in their judgements as the degree of ill-definedness of the domain is increasing. If agreement can be established at all, still resource requirements are high. Therefore, any gold-standard in an ill-defined domain is a compromise between the desirable and the possible.

For our purpose, we invited a tutor of the Logic Programming course to check all bugs reported by INCOM for every student solution, either confirming or rejecting it. In addition, the human tutor had

the opportunity to give a comment to each of the bugs. Finally, he had the option of adding general comments that are not specific to the presented bugs if, for example, he thought that crucial bugs have been missed.

For the evaluation, we only selected student solutions which had been classified as *understandable* by an human tutor, a category which had already been used for the evaluation of the intention analysis. *Not understandable* solutions are code fragments which could not be interpreted as a program (see an example in the Appendix). Table 7 summarises the results of system diagnoses on the student solutions.

Table 7
Evaluation of diagnostic reliability

Task	1	2	3	4	5	6	7	Average
Recall	0.948	1.000	1.000	0.901	1.000	0.981	0.952	0.969
Precision	0.843	0.875	1.000	0.891	0.974	0.953	0.952	0.927

The result of our evaluation of diagnostic reliability shows that with values between 0.901 and 1.000 recall is high. This indicates that the constraint set of the system is sufficient. We notice also that precision is always lower than recall. That means the diagnosis emphasises quantity more than quality. In particular, the low precision of Task 1 points to a particular weakness of the constraints relevant for this problem task.

We identified two main classes of bugs that had been marked as false diagnoses by the human tutor. The first one related to cases where the constraints were too rigid. For instance, INCOM did not allow a constant to be assigned to a variable using the operator **is**, like for example *X is 0*, although 0 is a valid arithmetic expression. Therefore, the human tutor considered this kind of bug detected by INCOM not acceptable. The second class of bugs which had been interpreted as false diagnosis by the human tutor occurred when two arguments of a subgoal were swapped. In this case, the system detected several low level bugs: for example superfluous coreference between two variables, or a coreference between two variables is missing. The human tutor expected a more compact feedback like *"Two argument positions A and B are swapped"* instead of many bugs considering coreferences between variables. To remove this deficiency of INCOM, more higher level constraints need to be specified.

Of course, the method we used to determine the gold standard based on actual system diagnoses creates a strong bias towards these error interpretations. If, for example, a solution is erroneous because of a swapped argument position, the diagnosis reports a number of constraint violations, which in principle are all justified but unnecessarily detailed. It seems, therefore, that the precision of the system's diagnosis is too optimistic.

The full set of data used during our evaluation are available from the homepage of our project (http://nats-www.informatik.uni-hamburg.de/view/INCOM/Dokumentation).

RELATED WORK

To check whether the diagnostic results of INCOM are competitive, we compared them with results from PROUST (Johnson, 1990), PITS (Looi, 1991) and Hong's Prolog tutor (Hong, 2004) with respect to intention analysis and diagnostic reliability because those systems provide similar difficult problem tasks (Rainfall problem, List reversal). In PROUST, programming plans are used to perform intention-based diagnosis of errors in PASCAL programs, PITS follows an algorithm-based approach and Hong's

Prolog tutor applies a transformation technique to diagnose errors in Prolog programs. These systems have been evaluated based on the following measures: 1) the percentage of programs whose solution strategy is identified correctly, 2) the percentage of correctly recognized (not recognized) bugs, and 3) number of false alarms, which are bugs detected by the systems but not expected by a human tutor. Note that for these systems no learning benefits have been reported yet. We compare INCOM with these three systems by calculating the Recall and Precision measures based on the stastistics reported in the corresponding literature and under the assumption that the gold standard of those systems has been specified comparatively.

The comparison in Table 8 shows that PROUST is superior with respect to intention analysis but its Precision is the lowest. Hong's Prolog tutor achieves the highest Precision, however, at the cost of a low Recall. Overall, INCOM combines an acceptable quality of intention analysis with a high diagnostic accuracy compared to the other systems.

Table 8
A comparison of the diagnostic accuracy

System	Intention Analysis	Precision	Recall
INCOM	87.9%	0.9266	0.9688
PROUST	96%	0.8787	0.8143
PITS	80%	0.9580	0.9913
Prolog tutor	80%	1.0000	0.6886

CONCLUSIONS

CBM is a relative new and promising approach to domain and student modeling. Instead of capturing the solution process as an admissible sequence of problem solving steps as model-tracing does, it specifies the properties that a correct or good solution should have. From this characteristics Lynch and colleagues (Lynch, Ashley, Aleven, & Pinkwart, 2006) draw the conclusion that CBM is pedagogically more appropriate for ill-defined domains. They also, however, admit that it remains an open question "*whether a constraint-based tutor can provide all of the feedback that is appropriate*."

In this paper we have shown that a highly informative and reliable feedback can be produced even for domains with a mild degree of ill-definedness by explicitly modelling solution strategies and high-level programming concepts. For this purpose we enhanced the CBM approach with constraint weights. This extension helps us to predict the strategy used by the student, to choose among competing correction proposals, and to reduce the complexity of individual constraints without making them too general. Still, the strategy-oriented diagnosis strongly relies on having a complete enumeration of possible solution strategies available.

Semantic constraints and transformation rules provide flexible means for mapping generalized solution strategies to a student solution. Together they span a solution space that is large enough to accommodate most of the correct *and* buggy solutions encountered in a real learning scenario, while at the same time producing the desired feedback in case of errors with a sufficiently high degree of reliability.

One of the major advantages of the constraint-based approach, on which its potential for dealing with ill-defined domains crucially relies, is its ability to provide useful feedback even in situations where no complete model of the domain is available. In such cases, errors might remain undetected, but the

much worse problem of blaming the student for the limitations of the model can be avoided to some degree. This advantage, however, can only be made productive, if constraints do not need to refer to complete enumerations of the available alternatives as the semantic table in our approach does. As long as a simple enumeration of alternatives cannot be avoided, CMB is effectively equivalent to the model-tracing approach in defining the space of correct solutions and any expectations about its potential to deal with ill-defined domains will remain an illusion. To overcome this problem, it would be necessary to formulate strategy-independent requirements for a solution, for example, based on a formal specification of the task and a model of the semantics of the programming language. That, however, remains a dream and a challenge for future research.

ACKNOWLEDGEMENTS

We are grateful to Thomas Kopinski for helping us to assess the system's diagnosis.

APPENDIX

A programming problem: Salary

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 creates a new salary list according to following rules: 1) Salary less equal 5000 Euro will be raised 3%; 2) Salary over 5000 Euro will be raised 2%. The new salary list would be: [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.

An example of a not understandable student solution

```
gehalt([Name, Gehalt|Rest], [N2,G2|R2]):- Gehalt<=5000, gehalt(Rest,...)
```

REFERENCES

Aleven, V. (2003). Using background knowledge in case-based legal reasoning: a computational model and an intelligent learning environment. *Artificial Intelligence*, 150(1-2), 183–237.

Anderson, J. & Reiser, B. (1985). The Lisp tutor. *Byte*, 10, 159–175.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4, 167–207.

Ashley, K. D. (2000). Designing electronic casebooks that talk back: The CATO program. *Jurimetrics*, 40(3), 275–319.

Bonar, J. G. & Cunningham, R. (1988). Bridge: Tutoring the programming process. In J. Psotka, L. D. Massey, and S. A. Mutter (Eds.) *Intelligent Tutoring Systems: Lessons learned* (pp. 409-434). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Goldin, I. M., Ashley, K. D., & Pinkus, R. L. (2001). Introducing PETE: computer support for teaching ethics. In *Proceedings of the 8th international conference on Artificial intelligence and law* (pp. 94-98). New York, NY: ACM.

Hong, J. (2004). Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies*, 61(4), 505–534.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence*, 42(1), 51–97.

Kodaganallur, V., Weitz, R., & Rosenthal, D. (2005). A comparison of model-tracing and constraint-based intelligent tutoring paradigms. *International Journal of Artificial Intelligence in Education*, 15(2), 117–144.

Kodaganallur, V., Weitz, R., & Rosenthal, D. (2006). An assessment of constraint-based tutors: A response to Mitrovic and Ohlsson's critique of "A comparison of model-tracing and constraint-based intelligent tutoring paradigms". *International Journal of Artificial Intelligence in Education*, 16, 291–321.

Le, N.-T. (2006). Using Prolog design patterns to support constraint-based error diagnosis in logic programming. In K. Ashley, V. Aleven, N. Pinkwart, and C. Lynch (Eds.) *Proceedings of the Workshop on Intelligent Tutoring Systems for Ill-Defined Domains at the 8th International Conference on Intelligent Tutoring Systems* (pp. 38-46). Jhongli (Taiwan), National Central University.

Le, N.-T. & Menzel, W. (2006). Problem solving process oriented diagnosis in logic programming. In R. Mizoguchi, P. Dillenbourg, and Z. Zhu (Eds.) *Proceedings of the 14th International Conference on Computers in Education* (pp. 63-70). Amsterdam, The Netherlands: IOS Press.

Le, N.-T. & Menzel, W. (2008). The coverage of error diagnosis in logic programming using weighted constraints - the case of an ill-defined domain. In D. Wilson and H. C. Lane (Eds.) *Proceedings of the 21st International FLAIRS Conference, Special Track on Intelligent Tutoring Systems* (pp. 421-426). AAAI Press.

Looi, C.-K. (1991). Automatic debugging of prolog programs in a prolog intelligent tutoring system. *Instructional Science*, 20, 215 – 263.

Lynch, C. F., Ashley, K. D., Aleven, V., & Pinkwart, N. (2006). Defining ill-defined domains; a literature survey. In K. Ashley, V. Aleven, N. Pinkwart, and C. Lynch (Eds.) *Proceedings of the Workshop on Intelligent Tutoring Systems for Ill-Defined Domains at the 8th International Conference on Intelligent Tutoring Systems* (pp. 1-10). Jhongli (Taiwan), National Central University.

Martin, B. (2001). *Intelligent Tutoring Systems: The Practical Implementation Of Constraint-based Modelling*. Ph. D. thesis, University of Canterbury.

Mitrovic, A. (2002). NORMIT: A web-enabled tutor for database normalization. In *Proceedings of the International Conference on Computers in Education* (pp. 1276-1280). Washington, DC: IEEE Computer Society.

Mitrovic, A., Koedinger, K. R., & Martin, B. (2003). A comparative analysis of cognitive tutoring and constraint-based modeling. In P. Brusilovsky, A. T. Corbett, and F. de Rosis (Eds.) *Proceedings of the 9th International Conference on User Modeling* (pp. 313-322). Berlin/Heidelberg: Springer-Verlag.

Mitrovic, A. & Ohlsson, S. (2006). A critique of Kodaganallur, Weitz and Rosenthal, "A comparison of model-tracing and constraint-based intelligent tutoring paradigms". *International Journal of Artificial Intelligence in Education*, 16, 277–289.

Mitrovic, A., Suraweera, P., Martin, B., & Weerasinghe, A. (2004). DB-suite: Experiences with three intelligent, web-based database tutors. *Journal of Interactive Learning Research (JILR)*, 15(4), 409–432.

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

Pennington, N. (1987). Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway (Eds.) *Emperical studies of programmers: 2nd workshop* (pp. 100-113). Norwood, NJ: Ablex Publishing Corp.

Pinkwart, N., Aleven, V., Ashley, K., & Lynch, C. (2006). Toward legal argument instruction with graph grammars and collaborative filtering techniques. In M. Ikeda, K. Ashley, and T. W. Chan (Eds.) *Proceedings of the 8th International Conference on Intelligent Tutoring Systems* (pp. 227-236). Berlin: Springer Verlag.

Pintrich, P. R., Berger, C. F., & Stemmer, P. M. (1987). Students' programming behaviour in a Pascal course. *Journal of Research in Science Teaching*, 24(5), 451–466.

Van Rijsbergen, C. J. (1979). *Information Retrieval* (2 ed.). London: Butterworth-Heineman.

Simon, H. A. (1973). The structure of ill structured problems. *Artificial Intelligence*, 4(3), 181–201.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J. A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., & Wintersgill, M. (2005). The andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15(3), 147–204.