INCOM: A WEB-BASED HOMEWORK COACHING SYSTEM FOR LOGIC PROGRAMMING

Nguyen-Thinh Le and Niels Pinkwart
Clausthal University of Technology
Germany

ABSTRACT

Programming is a complex process which usually results in a large space of solutions. However, existing software systems which support students in solving programming problems often restrict students to fill in pre-specified solution templates or to follow an ideal solution path. In this paper, we introduce a web-based homework coaching system for Logic Programming (INCOM) which allows students to develop a Prolog predicate in an exploratory manner, i.e., students are allowed to explore a solution space by themselves. As a tutoring model, this system supports students in two stages: a task analysis prior to an implementation phase. To model the solution space for a Logic Programming problem, a weighted constraint-based model was deployed in INCOM. This model consists of a set of constraints, constraint weights, a semantic table, and several transformation rules. The contribution of this paper is two-fold: it reviews a number of tutoring systems for learning programming which have not been considered in other surveys so far, and it presents an effective exploratory homework coaching system for Logic Programming.

KEYWORDS

Intelligent Tutoring Systems for programming, Logic Programming, weighted constraint-based model.

1. INTRODUCTION

Programming is a subject which is considered difficult by many students (Matthíasdóttir, 2006). To address programming novices' difficulties, several types of educational software systems have been devised, e.g.: programming environments, debugging aids, intelligent tutoring systems (ITS), intelligent programming environments, visualization and animation systems, and simulation environments (Deek & McHugh, 1998; Gómez-Albarrán, 2005). Among these types of systems, ITS which deploy AI techniques to support students solve programming problems as they advance and to develop their programming skills are rarely found in literature. The scarcity of successful ITS for programming can be explained by the fact that teaching programming is a difficult task even for a human tutor. Jenskins (2002) suggested that programming should be taught by persons who can teach programming, not by people who can program. In addition, diagnosing errors in a student's program is a challenge for a computational system due to the large space of possible correct (and incorrect) implementations. To be able to provide appropriate feedback to a student's program, a system must hypothesize the student's intention correctly. Otherwise, feedback is useless or even misleading. Two questions arise: Which educational approach should a tutoring system apply to support students in learning programming? How can a tutoring system identify errors in a student's program and give appropriate feedback according to the student's intention?

To address these two questions, in this paper, we introduce INCOM, a web-based homework coaching system for Logic Programming. INCOM supports a two-stage coaching strategy. First, the system requires students to analyze a problem specification and to represent results of their task analysis in form of a predicate signature. On the second stage, the system asks students to input a Prolog program in a free-form manner so that the requirements of the programming problem and the specified predicate signature are satisfied. To model a large solution space for each programming problem, we propose to apply the weighted constraint-based model (Le & Pinkwart, 2011).

2. TUTORING PROGRAMMING

In this section, we summarize findings about difficulties of programming novices and review existing ITS for programming. Then, we describe INCOM, a system which is intended to help students solve homework problems in Logic Programming.

2.1 Difficulties of programming novices

Novices' difficulties in learning programming are diverse. Here, we consider only some essential findings agreed among researchers. First, programming is a subject which requires a combination of both surface and deep learning styles. Deep learning means, students have to concentrate on gaining an understanding of a topic while surface learning focuses on memorizing facts. Thus, programming cannot be learned solely from books as in other subjects, instead students have to learn programming by developing algorithms themselves to deepen their understanding (Lahtinen et al., 2005; Bellaby et al., 2003). However, many students are not aware of the combination of two learning styles and apply inappropriate study methodologies (Gomes & Mendes, 2007). Second, there exists a correlation between programming and mathematical skills. According to Byrne and Lyons (2001) and Pacheco et al. (2008), it is necessary that programming novices should have been equipped with appropriate level of mathematical knowledge, because logical and abstract thinking is required for developing algorithms. To relieve this difficulty, Jenskins (2002) suggested that programming courses should be provided after some courses (including Math) have been taught. Third, programming languages, which are intended for professional use in the industry, usually have complex syntax and programming concepts. Thus, they are considered inappropriate for programming novices (Gomes & Mendes, 2007). These languages impose a high cognitive load for students to memorize a new syntax in addition to programming concepts and techniques. A solution is using an easy-to-learn programming language (e.g., Pascal). However, a counter-argument might be the student's worry about poor job market chances if learning a language which is not commonly used in the industry. Finally, programming is a complex activity which includes several sub-processes: analyzing a problem specification, transforming the problem specification into an algorithm, translating an algorithm into a program code, and testing a program. According to Jenskins (2002), the first sub-process is the most difficult and crucial phase of programming, because a correct and efficient algorithm is the basis of a program. While a programming expert has a set of solution schemata for a certain problem specification, programming novices usually struggle in this phase.

2.2 A survey of tutoring systems for programming

There exist numerous attempts to devising educational software systems which support students learn programming. Deek and McHugh (1998) classified 29 educational software systems for programming into four classes: programming environments, debugging aids, ITS, and intelligent programming environments. Programming environments and debugging aids can be used to relieve the novices' difficulties of learning syntax and concepts of a new programming language. These systems allow students to experiment with specific features of a new programming language and to observe the process of compilation. ITS usually provide students with programming tasks and integrated course materials. They are intended to support both surface and deep learning styles. Intelligent programming environments combine functionalities of ITS and programming environments. Twelve years ago, the authors of this survey concluded that a large part of research has rather focused on developing systems to support novices learn syntax and concepts and that the novices' difficulties in solving programming problems have not gained much attraction. At that time only few tutoring systems for programming had demonstrated successful: e.g., LISP tutor (Anderson & Reiser, 1985), which had been used for several years in regular programming courses; and ELM, a tutoring system for LISP (Weber & Möllenberg, 1995) whose successor is ELM-ART which is still being used in curriculum for LISP and available on the Internet (http://art2.ph-freiburg.de/Lisp-Course). In addition, Deek and McHugh also pointed out that in 1998, many systems under review constricted the student's freedom by proving students with solution templates to be filled due to the restricted ability of the underlying error diagnosis approach and thereby narrow down the possibilities to develop creative solutions. Gómez-Albarrán (2005) extended the existing classification of educational software systems with three other types: 1) example-based environments: the systems exploit examples to support students solve new problems and this

type of systems can be considered a specific type of ITS; 2) visualization and animation systems: these systems facilitate the teaching activity by displaying the behavior of the code; and 3) simulation environments which reflect the program execution in an imaginary world. With a survey of nearly 20 systems, Gómez-Albarrán stated that most systems focus on less sophisticated and less intelligent approaches, i.e., automatic error diagnosis in programs is not supported. Here, in this paper, we survey recent systems which support learning programming and have not been considered the surveys mentioned.

To address the first difficulty of programming novices, ITS for programming have been developed to support problem solving in a specific programming language, for example JITS for Java (Sykes, 2005), C-tutor (Song et al., 1997), and SQL-Tutor (Mitrovic et al., 2004). JITS aims at providing feedback to grammar errors in small Java programs. For each problem, the exercise author needs to specify a problem statement, a program specification, and a solution template to be filled by the student. The system compiles the student's program and analyzes grammar errors based on a cognitive model represented in production rules. The system has been assessed to be enjoyable, beneficial, and useful by both students and professors. Unlike JITS, which just focuses on grammar errors in students' programs, the C-tutor is able to diagnose both semantic and grammar errors in a student's program according to the student's intention. The program analyzer of the system has been tested with real students and is able to analyze 93% of students' programs correctly. SQL is not a universal programming language due to the underlying simple machine model (in fact, a SQL query is used to select data from a database, but it cannot be used to process data like a program), the SQL-Tutor supports students to define a SQL query to retrieve the appropriate data. When solving a problem, the system presents the student with a structured solution template which consists of pre-specified ordered slots for SQL constructs to be filled. The system's knowledge is coded using constraints. For each problem, the exercise author specifies an ideal solution. The system uses constraints to compare the student solution with required components of the ideal solution. This system has been reported as being useful to help students improve their skills in defining SQL queries. While we cannot comment on the user interface of the C-Tutor (it is not described in the cited literature), JITS and SQL-Tutor have one thing in common: they provide students with solution templates where blank spaces have to be filled, and this kind of interface restrict students to explore different possible implementations.

There exists another type of ITS for programming which rather focus on providing curriculum lessons according to a student's knowledge level than support problem solving. For example, the system developed by Sierra et al. (2007) relies on the idea that a transfer can be made between two domains which share similar concepts. The system supports students who have been exposed to a programming language to learn new programming languages. In the cited paper, the system supports Perl, Java and C++. BITS (Butz et al., 2004), a web-based intelligent tutoring system for computer programming, rather focuses on helping students navigate through the course material.

The second difficulty of programming novices can potentially be solved by studying Math. To our knowledge, there exists no software system intended to support programming by enhancing student's Math knowledge. Usually, universities offer a Math course in the early semester of a study.

The third difficulty of programming novices that the syntax and concepts of a new programming language are complex to learn can be relieved by providing programming environments enriched with text editors which are able to detect syntax errors and to propose correct syntax immediately while coding. Programming environments of this type can be found in (Deek & McHugh, 1998; Gómez-Albarrán, 2005). In addition, several systems have been developed to focus on a certain commonly used API of a specific programming language. For example, jTutors (Dahotre, 2011) searches examples for Java APIs on the Internet and provides them to students. jTutors uses CTAT as a platform to author and deliver tutorials related to APIs. For a similar purpose like jTutors, MICA (Stylos & Myers, 2006) searches on the Internet appropriate API classes and methods given a description of the desired functionality. In addition, the system helps students with examples for class methods.

Addressing the fourth difficulty of programming novices, i.e., transforming a problem specification to an algorithm, few software systems have been developed. PROPL (Lane & VanLehn, 2005) is an ITS system which uses communication patterns to coach students to understand a given problem specification and to develop a pseudo-code algorithm by holding a conversation. The system focuses on the activities of analyzing a task and planning a solution. Students who used this system were frequently better at creating algorithms for programming problems and demonstrated fewer errors in their implementation.

We have reviewed eight educational software systems for programming. Among them, only three systems (JITS, SQL-Tutor, and PROPL) have been evaluated with respect to their learning effect. We can infer that

other systems are still prototypes (or that only unsuccessful attempts at finding learning benefits have been made that were not published). This can partially be explained by the fact that developing and seriously evaluating a tutoring system requires a huge amount of time. Although a lot of time has been invested in building ITS for programming, most of them have not been deployed widely. This is in consistence with the observation of Eitelman (2006). Especially ITS focusing on problem solving like SQL-Tutor and JITS provide students with solution templates to be filled. These systems have demonstrated useful, however, the user interface of these systems restricts students' ability to develop a program freely. The rationale for this is that diagnosing errors in a program is not an easy process and the error diagnosis approaches underlying the SQL-Tutor and JITS are limited. In the following, we introduce INCOM, a coaching system for Logic Programming, which allows students develop a program in a free-form manner. INCOM consists of two components: a two-stage coaching model and a domain model.

2.3 The two-stage coaching model of INCOM

The programming novices' difficulty of transforming a problem specification to an algorithm has been confirmed by a pilot study for Logic Programming (Le, 2011). The authors reported that among 632 incorrect solution attempts submitted by students, on average 27.75% of errors were due to false task analysis, i.e., students were not able to specify correctly the clause head of a predicate. As an approach to help students overcome this difficulty, INCOM applies a two-stage coaching strategy: students are required to analyze a problem specification and to reproduce the analysis in form of a predicate signature prior to coding a predicate.

2.3.1 First stage: Task analysis

Task analysis aims at developing an understanding of the given problem specification and constructing a mental representation of the task. We propose to help students understand a programming problem by requesting them to reproduce information and goals given in the problem specification in form of an adequate predicate signature which consists of five components: a predicate name, argument names, meaning, type, and mode of each argument. *A predicate name* is the identifier of a predicate to be implemented. *Argument names* serve as unique identifiers for the argument positions of the predicate. *Meaning(A_i)* represents the purpose of the argument position A_i. *Type(A_i)* represents the data structure for the argument position A_i. Actually, Logic Programming does not require specifying a data structure for variables, the computation is based on unification techniques. However, from a pedagogical point of view, it is useful to request the student to specify the data structure she intends to use at a particular argument position. Most frequently used data structures in Logic Programming are atom, list and number. Apart from these data structures, other terms can be classified as "arbitrary type". *Mode(A_i)* is the calling mode for the argument position A_i. For a given predicate whose number of argument position is greater than 0, each argument position can be specified to be in one of three calling modes: input (+), output (-), or indeterminate (?).

INCOM provides a user interface which requires student to analyze a programming task (Figure 1). The interface is divided into three parts. The upper part is for displaying the problem specification. The middle part is for specifying the signature of a predicate to be implemented. The five labels 1-5 in this part indicate the place to input the five components of a signature accordingly. Using the box with label 0, students can add more argument positions or submit the predicate signature for evaluation. System's feedback is shown on the bottom part. Specifying a predicate signature in this way, the student is free to place the position of each argument and to name the identifiers according to her understanding of a given problem specification. As long as the signature input from the student is not yet appropriate with respect to the problem, the system provides feedback indicating students to think about highlighted important information and goals in the problem specification. Requesting students to specify a predicate signature is consistent with recommendation of Logic Programming experts. Brna (2001) suggested learners of Logic Programming comment their code in order to indicate a predicate's intended usage. From a technical point of view, the task analysis stage not only encourages the student to practice analyzing tasks, but also provides valuable information which helps to make the subsequent error diagnosis more accurate, because through specifying a predicate signature, students indicate the meaning of each argument position.

Although analyzing a programming problem by specifying a signature prior to the implementation is a good programming practice, this approach has several these limitations. First, this coaching approach is not

able to cover all possible understanding problems. For instance, a problem specification may contain the noun “list" which can be used to model the data type for an argument position. In case the student does not know the concept of a “list” structure in Logic Programming, then coaching her to specify “list” as a data type for an argument position would not help her further. This kind of knowledge should have been acquired during lectures or from text books, but not during the stage of task analysis. In addition, it is not always possible to derive a unique data type for an argument position from a noun phrase if the noun phrase does not indicate a data type explicitly. For example, the noun phrase “*A pair of persons*” does not point to a specific data structure. Therefore, various data structures can be used. In such a case, the student is forced to use the predicate signature exactly as specified by the exercise author. Furthermore, the requirement to specify noun phrases for the argument positions of a predicate signature explicitly might easily render the problem specification look artificially, e.g., most exercise descriptions do not include the noun phrase which represents the result of a computation.

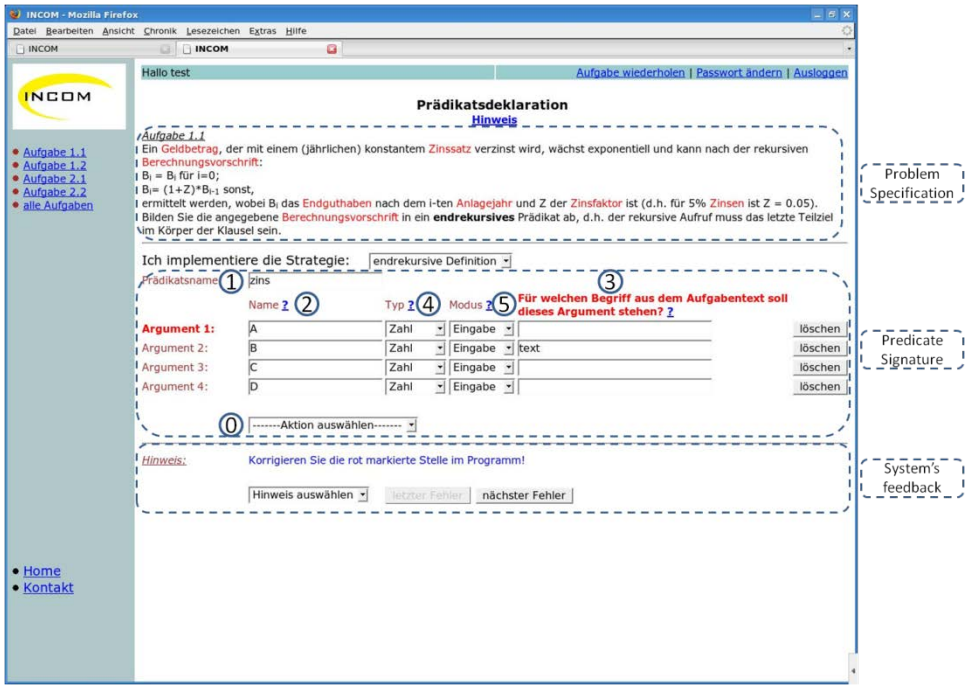

Figure 1: The interface for specifying a predicate signature

2.3.2 Second stage: Implementation

Once the student has provided an appropriate signature, the system guides her to the second stage where she is allowed to implement a predicate. The aim of this stage is to support students to develop a correct predicate for a given problem specification. For this purpose, the following requirements need to be satisfied. First, since students have specified a predicate signature, information about the agreed upon predicate signature should be available during the process of coding a predicate. Second, the implementation stage should allow students to input a solution in a free-form manner, because they should be able to explore a large space of possible solutions. Third, the system should provide feedback in consistence with the intention of students to help students develop a correct predicate.

INCOM provides a user interface for this implementation stage which fulfills these three requirements (Figure 2). The user interface includes four parts: 1) the top part for problem specification, 2) a display for the specified predicate signature, 3) a free-form input for solutions, and 4) the bottom part for system’s feedback. Although students are moderately restricted to a given solution structure (a clause always consists of a clause head and an (empty) clause body), this kind of layout still agrees with the requirement of designing a problem solving environment with free-form solution input. According to our survey in Section 2.2, most current ITS for programming have not considered this requirement. Using this user interface, first, the student needs to define necessary clauses by adding a new clause (Label 0). To be able to follow her intention, for each clause, the system asks her to additionally specify the type (recursive case, base case, and

non-recursive) of the clause she intends to implement (Label 1). Then, she is required to specify the clause head (Label 2) and a clause body (Label 3). After coding the predicate, the student chooses an action (Label 0) to submit her predicate for evaluation. If her predicate, including the main predicate does not fulfill the goals specified in the problem specification, the system provides feedback to improve the implementation. Feedback includes the location of and an explanation for the error is displayed on the bottom part of Figure 2.

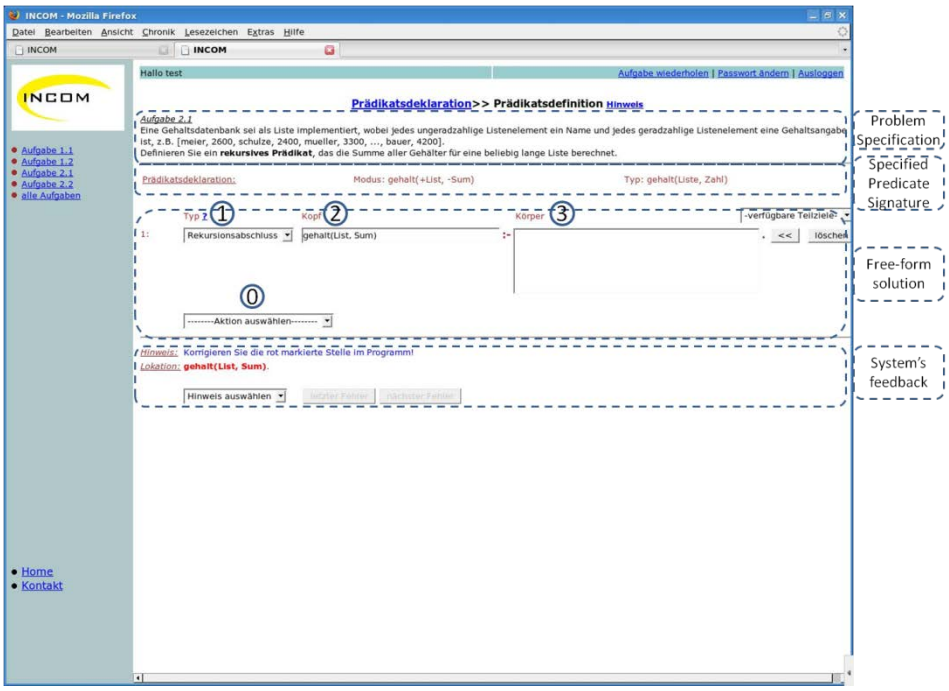

Figure 2: A user interface for developing a predicate in a free-form manner

2.4 The domain model of INCOM and its error diagnosis

The domain model of INCOM has been developed by applying the weighted constraint-based approach; see (Le & Pinkwart, 2011) for a detailed description. Here, we only outline the guiding principles of this approach which consists of four components. First, a semantic table is used to cover alternative solution strategies for a problem specification. Second, constraints are defined to check the semantic correctness of a student's predicate by comparing required components in the semantic table with the student's predicate. In addition, constraints are used to check general well-formedness of a predicate. Third, each constraint is associated with a weight value which indicates the importance of that constraint. The determination of the importance level for constraints resembles the assessment of examinations by a human tutor: if a solution contains more important components, then it receives a better mark. Fourth, to extend the coverage of possible solutions, transformation rules can be exploited to transform a code fragment without changing the semantics of the predicate, e.g. using commutative and distributive laws to transform an arithmetic expression. The process of error diagnosis generates hypotheses about the student intention by matching the student's predicate against alternative solution strategies in the semantic table iteratively. Hypotheses are evaluated using weighted constraints. The hypothesis which has violated constraints with "least" constraint weights is considered the solution strategy intended by the student. The violated constraints indicate errors in the student's predicate. In this process, constraint weights serve three purposes: 1) controlling the process of error diagnosis, 2) determining the student's intention, and 3) ranking feedback messages according to the severity of underlying errors. Enhancing constraints with weight values, Le and Pinkwart stated that the weighted constraint-based model is better suited to determine the student's intention than the classical constraint-based modeling technique without using weights.

3. EVALUATION

We conducted an evaluation of INCOM to determine whether students improve their skills in Logic Programming after having used the system. The study has been carried out during regular classroom hours, where normally students are expected to demonstrate their homework in the presence of a human tutor. This study took places in two sessions: in 2009 with 35 participants and in 2010 with 32 participants. In both sessions, a stable trend of the development of learning gains of the experimental group could be determined: the experimental group outperformed a control group (where students used a regular Prolog tool without feedback) by an effect size between 0.23 and 0.33 standard deviations (Le, 2011).

In addition to statistical results about the learning benefits of INCOM, based on a questionnaire, we asked students of the experimental group about their attitude towards the system. The questionnaire addressed the following issues: 1) the usability of the user interface, 2) the helpfulness of the two-stage coaching model, 3) the precision of error location, 4) the comprehensiveness of system's hints, 5) students' motivation, 6) the overall helpfulness of the system, 7) the confidence for solving similar tasks, and 8) using the system for homework. For each question, participants were asked to provide their opinion on a scale between 1 (very negative) and 5 (very positive). We accumulated the results of the questionnaire of two experiment sessions.

In addition, students from both control and experimental group were asked about the difficulty of given experiment exercises. 50% of the participants rated (very) difficult and 21% of them rated (very) simple.

Table 1: Students' attitudes towards INCOM (1: very negative; 5: very positive)

	Feature	m (s.d.)		Feature	m (s.d.)
1.	User interface	3.31 (0.7)	5.	Motivation	3.41 (1.3)
2.	Two-stage coaching	2.53 (1.1)	6.	System's helpfulness	2.57 (1.5)
3.	Error location	3.27 (1.1)	7.	Transferability	3.00 (1.3)
4.	System's hints	2.97 (1.5)	8.	Use for homework	2.47 (1.4)

Table 1 depicts the summary of students' attitudes towards the system INCOM. We can notice that students rated INCOM above average for the questions 1-7. Especially, the categories considering the user interface, the precise error location provided by the system, and the motivation of students using the system have highest ratings. With respect to the comprehensiveness of system's hints and the confidence of students in being able to solve future problems of the same type, a positive trend could also be identified. However, with respect to the two-stage coaching strategy, the helpfulness of the system, and the deployment of INCOM for homework, a clear positive answer could not be determined. Despite this cautious subjective assessment, the objective statistic results showed that at least some of them have made moderate learning gains.

We took the subjective results seriously and attempted to find reasons which led to a non-positive attitude towards the helpfulness and the deployment of INCOM in homework settings. First, students may be not aware about their learning progress while using the system because the time of using the system was too short (60 minutes). Second, they needed much time to become familiar with the functionality of the system. Five of total thirteen students' free comments (in addition to nine questions in the questionnaire) addressed this issue. In particular, participants of the experimental group spent between 17 minutes (in the 2nd session) and 21 minutes (in the 1st session) to analyze five programming tasks. This is a remarkable amount of time compared to the remaining time for the implementation stage during which the main coding activity takes place. Therefore, we can suspect that the first stage is one of the reasons why the usefulness of the system was not rated positive. Maybe feedback during this coaching stage was not sufficient because it could not elicit information (e.g, nouns used to represent a parameter position of a predicate) hidden in the problem specification as we discussed in Section 2.3.1. Third, the quality of feedback also plays an important role for the usefulness of the system. Although students rated the comprehensiveness of feedback above average (cf. Table 1), there was one participant who commented that system's feedback was of little use if it only explained the error without giving a recommendation how to correct a solution. Indeed, we assumed that students were able to derive a corrective action from an error explanation message.

4. CONCLUSION

In this paper, we have reviewed a number of educational software systems for learning programming and we have introduced the system INCOM intended to help students do homework in Logic Programming. The

novelty contributed by this system is that it supports students solve programming problems in an exploratory manner. INCOM provides a two-stage coaching model which requires students to analyze a task by specifying a predicate signature prior to coding the predicate. The domain model of the system has been developed by applying a weighted constraint-based approach. An evaluation study showed that the system was able to help students improve their ability of defining predicates for logic programs: students who used the system outperformed students that did not use it by 0.23 and 0.33 standard deviations. Positive attitudes of students after using the system could be identified. Students were motivated while working with the system and they felt confident to solve similar programming problems. However, a large number of students consider the two-stage coaching model as too restrictive, and the user interface needs to be improved. In future, we investigate the weighted constraint-based approach in the imperative programming paradigm.

REFERENCES

Anderson, J. R. & Reiser, B., 1985. The LISP tutor. *Byte 10*, pp.159–175.

Bellaby, G. et al., 2003. Why Lecture? *Proceedings of the 4th annual LTSN-ICS Conference*, pp.228-231.

Brna, P., 2001. Prolog Programming A First Course. Learning Unit, University of Leeds, UK.

Butz, C.J. et al., 2004. A web-based intelligent tutoring system for computer programming. *Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence.*

Byrne, P. & Lyons, G., 2001. The effect of student attributes on success in programming. *Proc. of the 6th Conference on Innovation and Technology in CS Education*, pp. 49-52.

Dahotre, A., 2011. jTutors: A web-based tutoring system for Java APIs, Master Thesis, Oregon State University. http://ir.library.oregonstate.edu/xmlui/bitstream/handle/1957/20562/jTutors_Aniket.pdf

Deek, F. P. & McHugh, J., 1998. A survey and critical review of tools for learning programming. *Journal of Computer Science Education*, Vol. 8, No. 2, pp.130–178.

Eitelman, S.M., 2006. Computer tutoring for programming education. *Proc. of the 44th annual SE. Regional Conf.,* USA.

Gomes, A. & Mendes, A.J., 2007. Learning to program-difficulties and solutions. *Proc. of Int. Conf. on Eng. Education.*

Gómez-Albarrán, M. 2005. The teaching and learning of programming: A survey of supporting software tools. The Computer Journal, Vol. 48, No. 2, pp. 130-144.

Jenkins, T., 2002. On the difficulty of learning to program. *Proc. of the 3rd annual Conference of LTSN-ICS*.

Lahtinen,. E. et al., 2005. A study of the difficulties of novice programmers. *Proc. of the 10th annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pp.11-18.

Lane, H. C. & VanLehn, K. 2005. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education*, Special issue on doctoral research in CS Education 15 (3), 183–201.

Le, N.T., 2011. Using weighted constraints *to* build *a* tutoring system *for* logic programming. Phd dissertation, University of Hamburg, http://ediss.sub.uni-hamburg.de/volltexte/2011/5141

Le, N.T. & Pinkwart, N., 2011. Adding weights to constraints in Intelligent Tutoring Systems: Does it improve the error diagnosis? Proc. of the *6th European Conference on Technology Enhanced Learning*.

Matthíasdóttir, A., 2006. How to teach programming languages to novice students? Lecturing or not? *Proc. of Int. Conference on Computer systems and Technologies*, pp. IV-13-1-IV-13-7.

Mitrovic, A. et al., 2004. DB-suite: Experiences with three intelligent, web-based database tutors. *Journal of Interactive Learning Research* 15 (4), 409–432.

Pacheco, A. et al., 2008. Mathematics and programming: some studies. *Proc. of Int. Conference on Computer Systems and Technologies*, pp. V.15-1-V.15-6.

Sierra, E. et al., 2007. A multi-agent intelligent tutoring system for learning computer programming. *Proceedings of CERMA Electronics, Robotics, & Automotive Mechanics Conference*, pp. 382-385, IEEE Computer Society Press.

Song, J.S. et al., 1997. An intelligent tutoring system for introductory C language course. *Journal of Computers & Education*, Vol. 28, No. 2, pp. 93-102.

Stylos, J. and Myers, B., 2006. MICA: A web-search tool for finding API components and examples. *Proc. of the Visual Languages and Human-Centric Computing*, pp.195-202.

Sykes, E., 2005. Qualitative evaluation of the Java Intelligent tutoring system. *Journal of Systemics, Cybernetics and Informatics*, Vol. 3, No. 5, pp.49-60.

Weber, G. and Möllenberg, A., 1995. ELM programming environment: a tutoring system for Lisp beginners. In Wender, K., Schmalhöfer, F., & Boecker, H.D. (eds.), *Cognition and computer programming*, pp. 373–408. Ablex Publishing.