Meaningful Learning in the Tutoring System for Programming

Tonci Dadic, Slavomir Stankov, Marko Rosic
Faculty of Natural Science, Mathematics and Education, University of Split
Teslina 12, 21000 Split
{tonci.dadic|slavomir.stankov|marko.rosic}@pmfst.hr

Abstract. Tutoring systems for programming helps students to understand features of target programming language, and develops their general problem solving skill. Our system guides novices in learning syntax and semantics of programming language, problem decomposition, program design and testing. The notional machine defined by programming language and its verbal description of instruction actions helps students to understand semantics of instructions. Advancement through the course material controlled by computer teacher supports connection of new concepts to the present student’s knowledge.

Keywords. Programming teaching, Intelligent Tutoring Systems, TEx-Sys model.

1. Introduction

Intelligent tutoring systems for programming are an interesting area of artificial intelligence application. Since early seventies, tens of systems have been designed and hundreds of papers published, related to procedural [10][14], functional [1], logical and object languages [4], including problem analyzes, program specification, planning [11] and coding.

In addition, researchers focused to adaptive navigation [3], visualization of program execution and intelligent debugging [7].

Regardless significant research effort in field, there are very few intelligent tutoring systems or integrated learning environments used regularly for teaching programming in a real classroom, and most of them can not be used outside the university where they were created [2]. Design of environment supporting efficient programming learning is a challenging domain with many open issues to explore [8].

In this paper we present our system for programming learning of procedural languages, focusing on notional machine defined by programming language and advancement through the course material. We discuss student’s evaluation and our experience of the system usage in classroom teaching.

2. Background and related work

Cognitive psychology finds an important role of meaningful learning in new information acquisition. The activation of present knowledge from learner’s long term memory helps assimilation of new knowledge. Mayer [12] elaborates how concrete, familiar models improve understanding of new technical information. He argues that verbal description of statements execution helps novices to understand programming language features.

Since 2006, when we presented prototype model of our system [5], it matured due experimental use in classroom. Our system controls students’ advancement through lessons, providing connection of new information with concepts they have learnt. We have introduced model of notional machine defined by programming language as a glass box that executes instructions and describes their actions by natural language.

LISP Tutor [1] diagnoses student’s knowledge applying model tracing, keeping him always on the path of model solution. PROUST [10] performs late semantic analyzes of programs compiled by traditional Pascal compiler. Our system distinguishes students performing well and those with difficulties. The first are allowed to experiment with program due late semantic analyzes, while later get help in problem decomposition immediately as they are faced with trouble.

3. Our system

Tutoring systems for programming helps students to understand features of target
programming language, and develops their general problem solving skill. Our system guides novices in learning syntax and semantics of programming language, problem decomposition, program design and testing. It is capable to teach different procedural languages. New language is introduced by grammar in Bachus-Naur form, while knowledge base is common, language independent, and system dynamically accommodate course material to specific language, subject of each course.

3.1. Client - server architecture

The system employs server – client architecture based on web services. Central knowledge base, implemented by semantic network holds concepts definitions and links. Courseware database is central as well, implemented by relational schema holding users, available courses, curriculums and links referring to concepts of knowledge base.

The architecture supports usage of client processing power in real time analyzes of student’s program as well as centralized knowledge and tools configuration settings.

![Figure 1. System client-server architecture](image)

3.2. System modules

Computer expert tries to understand student’s program, locate buggy code and diagnose the bug generators in students’ thinking model. Expert analyzes student’s source code and predicts possible paths to final solution in each point of program. Understanding of student’s program is based on two artificial intelligence techniques: Hartman’s decomposition [9] and Constraint satisfaction problem [15].

Computer teacher guides student from one lesson to other, and chooses programming tasks appropriate to his current knowledge, skill and learning ability. It observes student’s work, interrupting if syntax or semantic error is found.

Student model identifies current status of student’s knowledge, as well as his progress in concepts adoption over variety of tasks. In order to follow student’s progress, the layout approach, traditionally used to diagnose student’s knowledge is extended by sequences of error and help indicators. System updates student model database with data collected during his activity in tasks solving.

Tools module contains structured source code editor, compiler and abstract machine implementing model of programming language. Abstract machine supports program execution step by step, shows variables and describes actions of each instruction.

3.3. Teaching scenario

We have based the teaching scenario on facts that there exist distinguished categories of programming knowledge, and individual learners’ ability. Programming requires language syntax knowledge, understanding of single instruction semantics, program design and testing skills. Students might differ widely not only in knowledge they bring to course, but how they deal with difficulties. Perkins et al. [13] identifies two different learning styles: stoppers and movers. Stoppers tend to disengage from programming task at the first sign of trouble,
whereas movers will begin tinkering with a partial solution. Perkins finds that stoppers can become movers if they are encouraged to decompose problem and focus on simpler sub-problem only.

### 3.3.1. Teaching syntax

System watches source code editor character by character as student types. Delimiters (i.e. blank, new line) trigger syntax analyzes of piece of code taken from the start of program to the current cursor position. Whenever syntax error is found, student is interrupted with a message, for example:

*Syntax error found at row 3, column 12.*

If student fixes error on his own, without future system help, it is assumed like typing error. While student is not able to get error free instruction, system provides three level help:

1. **Found – expected:** expected tokens instead of found one are shown.
2. **Lesson launching:** the lesson explaining syntax of statement is displayed.
3. **Detailed help:** solution to prune error out is shown.

System deals with syntax errors immediately, in computer coach manner, keeping student's focus on single problem at a time. Student is stopped at concept he doesn’t know and has to learn. In opposite, error description delayed to compile time, could lead to accumulation of many errors and confuse novices.

### 3.3.2. Teaching statement semantics

Students’ understanding of single statement semantics is assessed by three separated techniques:

*Reformulation:* source code of program is presented, asking student to do small modifications, for example change output messages or number of repeated iterations.

*Transformation:* student has to rewrite given code using different statements. For example, student can transform repetition implemented by `if` and unwanted `go-to` to `while`, if knows semantics of all these instructions.

*Mental simulation of program execution:* student is asked to predict the sequence of specific variable values or program output. Dialog is used to get student’s answer and inform him about his findings. In case of incorrect answer, the new attempt with system directions is allowed. When correct answer is not achieved within three attempts, student may use programming language model, run program step by step and follow variable values.

### 3.3.3. Teaching program design skills

Design skills are the group of techniques used to combine language features in order to form a program. These include knowledge of stereotypic patterns of code that perform complex functions such as sorting. Programmers decompose the problem into component parts and plan how to combine them into solution as a whole.

Our system helps to design program comparing student's and expert's solutions of given task. Expert's solution is organized in a tree of goals and plans, where root represents the program and leafs are plans. The goal is tuple of unique identifier, hint, description and list of statements – plan. The process of program recognition maps student’s plan to expert’s plan. Success of mapping means that student has accomplished the sub-goal. Unsuccessful mapping is assumed as a semantic error of specific sub-goal. We apply different approaches in recognition of stopper’s and mover’s programs. Stopper's program (syntactically correct) is analyzed immediately as he changes code. We call it early semantic analyzes. Delay of student’s action or semantic error interrupts his work and arises three level help:

1. **Hint** – system gives direction how to realize the goal.
2. **Description** – the goal is explained in details.
3. **Plan** – code that accomplishes the goal is visible.

The system performs semantic analyzes of mover’s program in the moment he calls compiler to translate program. We call it late semantic analyzes. Program understanding is performed by application of constraint satisfaction problem. Broken constraints diagnose semantic errors. Early semantic analyzes wakes up stoppers and keeps them on the path of correct solution. With late semantic analyzes movers get opportunity to experiment with program finding correct solution on their own.

### 3.4. Notional machine

DuBoulay et al. [6] identifies two basic approaches in learning to interact with a
computer: the black box and the glass box. In the black box approach one puts in commands and data and gets the answer as if by magic. The mechanisms behind cover are hidden from the user, and the user is likely to assume that computers are just not understandable. Such users might memorize algorithms that solve one or other task, but are not able to understand what goes on inside the black box.

In glass box approach the user attempts to understand what is going on inside the computer running program. These actions can be described and understood. The important properties of model making hidden operations of language clearer to novices are: right level of details, simplicity and visibility.

Instructions of a program interact with memory locations – variables. Conditionals check variable values to make decision if some other instructions have to be executed, assignment changes variable values etc. Thus we implemented programming language model with properties:

- Student may trace program running instructions step by step.
- Segment of memory allocated to variables is displayed. It is easy to change value of any variable and check effect on current step.
- The system verbally describes actions of each instruction.

Example of an assignment depicts our model.

**Table 1. Program model**

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Variables</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>B</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Program**

\[ \begin{align*} A & := 1; \\ B & := A + 2; \\ \text{Write}(A, B); \end{align*} \]

Table 1 shows current memory segment allocated to variables A and B, as well as program segment running step by step.

The system describes next instruction to be executed as follows:

- Read value 1 from memory location 100, that is variable A. Value 1 of location remains unchanged.
- Read constant 2 from program text.
- Perform addition of values 1 and 2.
- Save result 3 to memory location 101, that is variable B. Previous value 0 of location is destroyed and can not be recovered ever.
- Move next instruction pointer to row 3.

Our model is adaptable to the current status of student’s knowledge. For example, if he knows scope and life time of variable, model shows appropriate information as well.

### 3.5. Advancement through the course

A course material is organized in chapters, sections and lessons. Each lesson explains a number of concepts. The course structure is represented by a tree, where course is mapped to root and lessons are leafs.

![Figure 2. Sample of course structure](image)

The learning process is identified by current lesson student deals with it, and his activity, that is textbook reading or task solving. Student starts with reading and clicks on the button when wishes to advance to the next stage of the course. The system responds displaying text of next lesson or switching student’s activity to task solving.

Switch to the task solving triggers system process that selects task-candidates appropriate to the current status of student’s knowledge. We describe each task by a set of concepts implemented by that task. The task is applicable candidate if all are true:

- The task implements at least one concept explained in current lesson.
- The task doesn’t implement any concept that hasn’t been explained in current or previous lessons.
- The task implements at least one concept that student doesn’t know by the student model.
We say, student knows specific concept if has solved several tasks implementing that concept.

**Algorithm:**

```plaintext
Select Tasks( T, L, E, K ) returns C: list of tasks
T: set of tasks associated to the course
P_i: not empty set of concepts implemented by i-th task.
L: set of concepts explained in current lesson
E: set of concepts explained in previous lessons
K: set of concepts student knows

O := L ∪ E;
C := ∅;
foreach ( Task T_i in T )
{
    if (P_i ∩ L = ∅ and P_i \ O = ∅ and P_i \ K ≠ ∅)
    {
        C.Append( T_i );
    }
}
sort list ( C, by tasks complexity );
return C;
```

The computer teacher chooses task by task from list of task candidates and asks student to solve them. Tasking starts from the middle of list sorted by level of task complexity. If student’s ability solving task is high, next task for him is picked from the middle of upper half of list. Task pointer is turned to the simplest task at top of list if student’s performance is low. When an error in student’s program is found system interrupts his work as we have explained in section 3.3.1.

Computer teacher analyzes current status of student’s knowledge and makes decision about his readiness to advance to the next lesson. As long as he is not found ready, some more tasks related to explained matter are presented.

Some students bring certain programming knowledge to the course, and want to skip simpler lessons. They can select lesson of their interest from the menu of course content. Advancement is allowed if student model proves that student knows all required concepts. Required-concepts-set contains all concepts explained in lessons that precede selected one.

If student hasn’t proved knowledge of some concepts yet, computer teacher will find a list of tasks implementing those concepts. Students that are not able to solve these tasks may not advance to lesson they selected.

Advancement through the course implemented by our system insures that student possess knowledge that can be activated in order to assimilate new information.

4. The system evaluation

During winter semester 2007/08 we have used our system in course of programming at Faculty of Natural Science, Mathematics and Education. Three groups of 29 students in total have learnt programming concepts using QBASIC. They followed teaching scenario described in section 3.3. 16 (55%) of our students hadn’t got any programming experience before course, while 13 (45%) had learnt programming in high school from ½ to 4 years. The later students had in average 3 semesters of programming experience in Pascal, C/C++ or QBASIC.

At the end of course students filled up a questionnaire based on 5 point Likert’s scale concerning their motivation to learn programming, impression about system and it usefulness in learning.

**Table 3. The system usefulness in learning**

<table>
<thead>
<tr>
<th>Point</th>
<th>Answers</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Very Harmful</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>2. Harmful</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>3. Can’t decide</td>
<td>3</td>
<td>10.3</td>
</tr>
<tr>
<td>4. Useful</td>
<td>4</td>
<td>13.8</td>
</tr>
<tr>
<td>5. Very Useful</td>
<td>22</td>
<td>75.9</td>
</tr>
</tbody>
</table>

We conclude that our system has achieved its main goal of usefulness in learning as 26 (89.7%) students found that system helped them to acquire the programming knowledge.

Students were asked to grade at 7 point Likert’s scale how much each of components helped them in learning. *(Point 1 is very harmful, 4 is neutral and 7 very useful.)*

**Table 4. Grade of usefulness**

<table>
<thead>
<tr>
<th>Item</th>
<th>Median</th>
<th>Mean</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notional Model</td>
<td>7</td>
<td>6.2</td>
<td>4.7</td>
</tr>
<tr>
<td>Advancement</td>
<td>5</td>
<td>5.0</td>
<td>1.7</td>
</tr>
<tr>
<td>Syntax help</td>
<td>7</td>
<td>6.2</td>
<td>4.7</td>
</tr>
<tr>
<td>Semantic help</td>
<td>7</td>
<td>6.7</td>
<td>6.7</td>
</tr>
</tbody>
</table>

Some students found advancement is very harmful, because they would prefer to select lessons on their own and ignore planed sequence of concept adoption. As we expected, 20 students graded semantics help very useful and 9 useful.

5. Conclusion

Our tutoring system for learning programming helps novices to understand what
goes on behind cover of programming language. Notional machine provides program execution step by step, describes instruction actions and makes variable values accessible. Controlling advancement through the course material it encourages students to learn systematically, from simpler to more demanding lessons. Switching from declarative knowledge presentation to task generation, the system provides application of explained programming concepts in problem solving.

Our system distinguishes students’ learning style providing immediate feedback to stoppers and delayed to movers, whereas LISP Tutor [1] always provides immediate feedback. PROUST [10] and INTELLITUTOR [14] semantically analyze students’ programs when they are completed, so that their intervention may be too late for many novices and early misconceptions have the chance to propagate throughout the whole program.

The evaluation of our system in real classroom argues its helpfulness in programming learning.

6. Acknowledgements

This work has been carried out within projects 177-0361994-1996 Design and evaluation of intelligent e-learning systems, funded by the Ministry of Science and Technology of the Republic of Croatia.

7. References