An ITS for Programming to Explore Practical Reasoning

Leliane Nunes de Barros, Karina Valdivia Delgado and Andréia Cristina G. Machion
{leliane, kvd ,amachion}@ime.usp.br
Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP),
Rua do Matão, 1010, Cidade Universitária – 05508-090
Phone Number: (011)30919878 - São Paulo, SP, Brasil

Abstract: Research on cognitive theories about programming learning suggests that experienced programmers solve problems by looking for previous solutions that are related with the new problem and that can be adapted to the current situation. On the other hand, an apprentice who does not have any previous programming experiences in mind can only appeal to the sentences of the language that he had learned so far. Inspired by these ideas, programming teachers have developed a pattern based programming instruction. In this model, learning can be seen as a process of pattern recognition, which compares experiences from the past with the current situation. In this work, we present a programming environment in which a student can program using a set of pedagogical patterns, i.e., elementary programming patterns recommended by a group of teachers. In this environment, while the student is editing a program, he can select and insert patterns in order to satisfy subgoals of a given problem. After having a compiled program, the student can submit it to a diagnosis system for detection of (i) possible errors and/or (ii) student's misconceptions on the use of patterns. Finally, in order to propose further extensions to the pattern based programming education, we analyze the programming reasoning in terms of a BDI architecture for rational agents in AI.

1 Introduction

Writing a program for a novice involves many difficulties and the attempts to deal with multiple impasses all at once, can make this task even worse. Research on programming psychology points out two challenges that a novice programmer has to handle:

1. **learning a new programming language:** the student has to learn and memorize the syntax and semantics of a new programming language;
2. **learning how to solve problems to be executed by a computer:** the student has to learn how to translate a solution, that she probably already knows how to solve by hand, to a program for the computer to execute. A good example could be how to construct a program to solve quadratic equations.

Although a programming language has a lot of details, the first challenge is not the most difficult part. Evidences show that learning a second language is, in general, easier. A hypothesis is that the student has already acquired abilities to solve problems using the computer, which is the common skill to learning different programming languages.

Related to the second challenge, research on cognitive theories about programming learning has shown evidences that experienced programmers store and retrieve old experiences on problem solving that can be applied to a new problem and can be adapted to solve it. On the other hand, a novice programmer does not have any real experiences but the primitive structures from the programming language he is currently learning [JS84]. Inspired on these ideas, one strategy to teach how to program is to present small programming pieces, instead leaving the student to program from scratch. That is the proposal of the *Pedagogical Patterns* Community: a group of experienced programming educators engaged in recommending programming pieces for novices also called *pedagogical programming patterns* or *elementary programming patterns*. Supposing that students who learned elementary programming patterns will in fact construct programs with them, i.e., with known pieces of code, an Intelligent Tutoring System (ITS) could take a number of advantages from this teaching strategy, such as:

- to allow the tutor to establish a dialogue with the student in terms of problem solving strategies, through the pedagogical pattern documentation, as it was done in PROUST [John90];
- to enable the tutor module for diagnosing the student's program, to reason about the pedagogical patterns in a hierarchical fashion, i.e., to detect program faults in different levels of abstraction.

In this paper, we present a new Eclipse IDE for programming learning based on Pedagogical Patterns that has been developed as part of Eclipse IBM/IME project (2004 grant). We also show an extension of the IDE with a diagnosis system, using a *Model Based Diagnosis* method, to detect errors in the student program in terms of: (1) the wrong use of the language sentences and; (2) the wrong selection and decomposition of Pedagogical Patterns. Finally, we discuss how the above reasoning model, i.e., the retrieval of programming patterns to solve a new problem, can be related with a *BDI* model for practical reasoning of rational agents in Artificial Intelligence and be useful to new investigations on programming learning strategies.

2 Pedagogical Programming Patterns in the Classroom

Pedagogical programming patterns, also called *elementary programming patterns*, are recommended solutions for common problems described in a way to facilitate their reuse. Patterns are simple, synthetic and recommended by researchers on programming teaching for novices [PC03]. A pattern relates a problem to a solution and provides information about the context that it can be applied. Its potential use in programming teaching has been explored by the pedagogical patterns community. Programming patterns are available in the Web for C, C++ and Java languages [Wal01], including: selection patterns [Ber99], repetition patterns [AW98] and others [Bri02]. Porter and Calder [PC03] suggest a process to employ programming patterns in the classroom and Proulx [Pro00] created a first Computer Science course based on these patterns.

Table 1: Programming Patterns Examples.

pattern name	**use / application**	**syntax**

Loop with Sentinel	You want to repeat a set of actions while a condition is true. In general, the set of actions is related to the processing of a sequence of elements or numbers. The amount of elements is unknown but the end of the sequence is indicated by a sentinel value. The elements can be read or generated.	'INITIALIZATIONS'
		'SENTINEL VARIABLE INITIALIZATION'
		while ('SENTINEL VARIABLE CONDITION')
		{
		  'READ/GENERATE A SEQUENCE ELEMENT'
		  'PROCESS ELEMENT'
		  'UPDATE SENTINEL VARIABLE'
		}

Counting Loop	You want to repeat a set of actions a determined number of times. In general, the set of actions is related to the processing of a sequence of elements or numbers. The number of elements must be known. The elements can be read or generated.	'INITIALIZATIONS'
		for ('COUNTER INITIALIZATION'; 'COUNTER CONDITION'; 'UPDATE COUNTER')
		{
		  'READ/GENERATE A SEQUENCE ELEMENT'
		  'PROCESS ELEMENT'
		}

Programming patterns can help novice programmers in two ways:

1. to learn general strategies (in a higher abstraction level);
2. to retrieve the syntax and the use of a programming language, once its documentations include a program, which is an example for that pattern application.

On the other hand, programming patterns can help a human tutor to: (a) recognize the student's intentions; (b) establish a better communication with the student, since they provide a common vocabulary about general strategies for programming problem solving.

pattern name *Loop with Sentinel*

use/application You want to repeat a set of actions while a condition is true. In general, the set of actions is related to the processing of a sequence of elements or numbers.
The amount of elements is unknown but the end of the sequence is indicated by a sentinel value.
The elements can be read or generated.

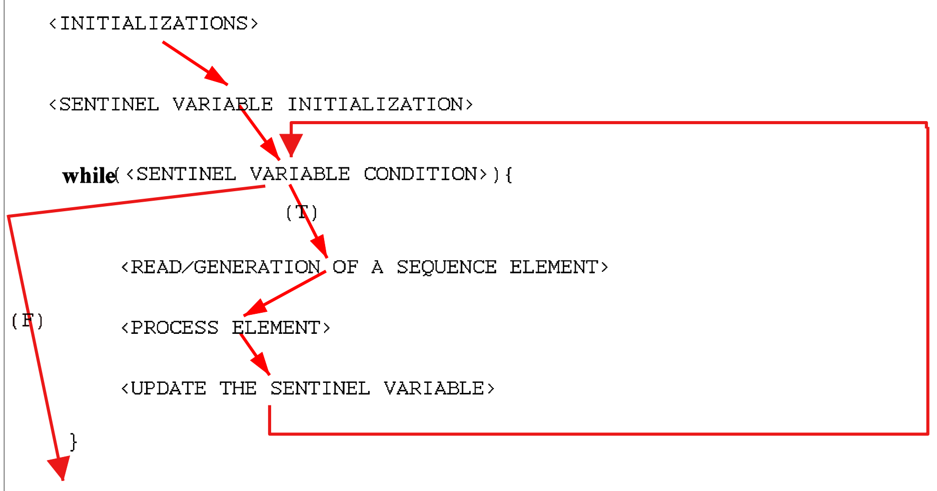
dependent patterns *Integer Variable Declaration, Set of actions*

syntaxe

```
  <INITIALIZATIONS>
  <SENTINEL VARIABLE INITIALIZATION>
  while  (<SENTINEL VARIABLE CONDITION>){
       <READ/GENERATION OF A SEQUENCE ELEMENT>
       <PROCESS ELEMENT>
       <UPDATE THE SENTINEL VARIABLE>
  }
```

The set of actions is composed by
<READ/GENERATION OF A SEQUENCE ELEMENT>, <PROCESS ELEMENT> and
<UPDATE THE SENTINEL VARIABLE>

semantics Variables and sentinel initialization as true condition is executed just once. After that, the sentinel condition is verified: (1) a sequence element is read/generated, (2) processed and, eventually, (3) the sentinel variable is updated.
Next, the sentinel variable condition is verified again.
The process stops when the sentinel variable condition becomes false. For that, the sentinel variable update and condition must be stated correctly, otherwise, the program will fall in a "infinite loop".



example: Write a program to read a sequence of integers, finished by zero and to calculate their sum.

```
sum=0;
printf("Input an integer: ");
scanf ("%d", &number); /*read the first number */
while (number != 0) {
        sum = sum + number;
        printf ("Input an integer: ");
        scanf("%d", &number);  /*read next number*/
}
```

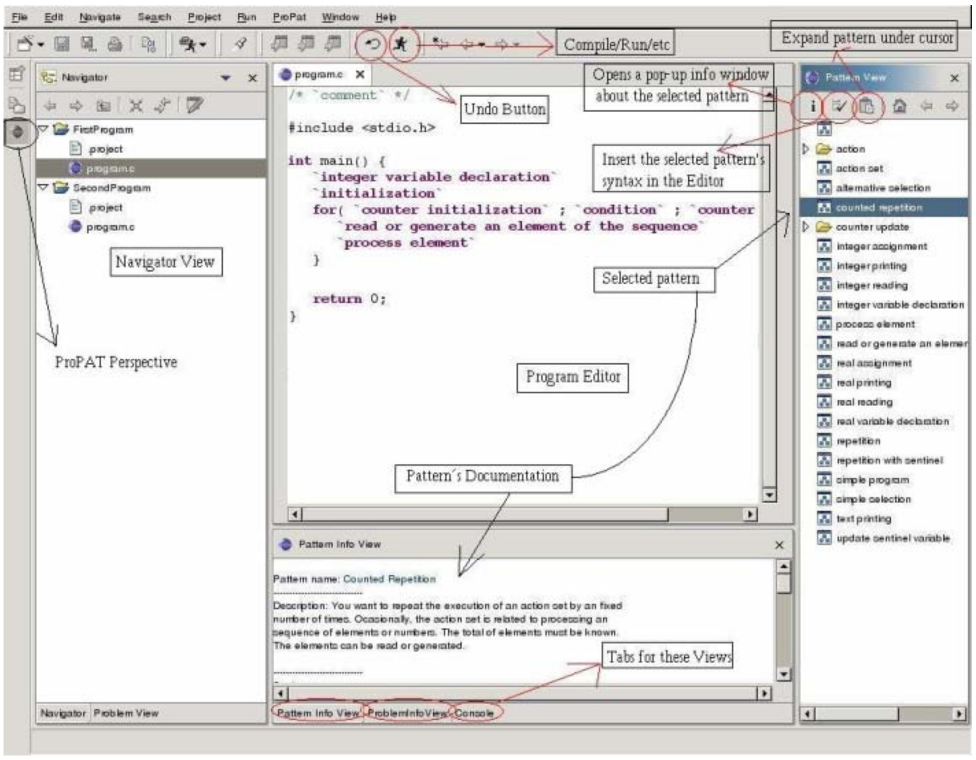
Figure 1: Loop with Sentinel Pattern.

Some pattern examples are shown in Table 1, where we give a small part of the documentation of two loop patterns: *Sentinel Loop* and *Counting Loop.* Notice that they correspond to distinct strategies commonly used to solve sequence treatment problems, as it is stated in the use/application column of the table. A student that uses one strategy when the other one is more naturally applied has difficulties to accomplish the program solution. In the syntax of the patterns (third column) the words between quotation marks are called *metadata.* Metadata are used to indicate to the student a part of the code that has to be instantiated with a new pattern or C code, typed by herself. Figure 1 shows the Loop with Sentinel example in more details.

3 ProPAT

ProPAT is a programming learning environment using pedagogical patterns, built as an Eclipse plug-in and is part of an IME-IBM project. ProPAT provides an IDE for a first computer science course, i.e., an IDE for programming to novices. In this environment, the student can choose a programming exercise and to construct a solution by selecting and adding Programming Patterns into the Eclipse editor. ProPAT also allows a teacher to add new pedagogical programming patterns, new exercises and bench tests.

This project was first developed for C language. Some ProPAT characteristics were inherited from the original Eclipse CDT plug-in while others were specially developed (Figure 2), such as:



```
/* `comment` */

#include <stdio.h>

int main() {
   `integer variable declaration`
   `initialization`
   for( `counter initialization` ; `condition` ; `counter
      `read or generate an element of the sequence`
      `process element`
   }

   return 0;
}
```

Pattern Info View

Pattern name: Counted Repetition

Description: You want to repeat the execution of an action set by an fixed number of times. Occasionally, the action set is related to processing an sequence of elements or numbers. The total of elements must be known. The elements can be read or generated.

Figure 2: ProPAT student perspective.

- **Student Perspective:** where the student can choose the exercises and develop solutions through patterns selection or even write their own code.
- **Teacher Perspective:** used by the teacher to specify new exercises and patterns that will be available to students.
- **Editor *View*:** a program editor in which the student can automatically include patterns by replacing metadata or typing a new line of code. The inclusion of patterns in the place of a metadata is constrained by a general taxonomy of metadata and patterns. Therefore the

implementation of this *view* involves the detection of a number of forbidden pattern selections (e.g.: 'initialization' can not be replaced by a 'print' pattern).

- **Navigator *View*:** allows the navigation between projects and their respective files.
- **Pattern *View*:** shows a pattern list from where the student can select a pattern, either to visualize its documentation or to insert it in the editor.
- **Pattern Info *View*:** shows the selected pattern documentation.
- **Problem *View*:** allows browsing in the exercises list, grouped by categories.
- **Problem Description *View*:** shows the selected exercise description.
- **Message Console**: shows the compilation messages.

Our proposal is to add a diagnosis module to ProPAT in order to detect errors in the student program (after it has been compiled with no syntax errors). In the next section we show how a well known model based diagnosis technique for physics systems [Ben93] can be used to programs [CMW00].

4 Diagnosis

The basic idea of model-based diagnosing programs is to derive a component model directly from the program and from the programming language semantics. This model must identify components, connections, the program structure and the system description. Similar to diagnosis of physical devices, the system description, in this case, is the student program behavior which reflects its errors. The observations are the incorrect outputs in the different points of the original program code. The predictions are not made by the system, but by the student and therefore this is the situation where the student communicates her programming goals to the tutor.

4.1 A review of Reiter's original algorithm

Reiter [Reiter87] proposed a diagnosis algorithm (for faulty systems in general) that computes all minimal hitting sets for a family of components sets **F** improved later by Greiner [Greiner89]. The algorithm generates an acyclic graph in which nodes are labelled by sets and arcs are labelled by elements of the set. The idea is that for each node labelled by a set **S**, the arcs leaving from it are labelled by the elements of **S**.

Let **H(n)** denote the set formed by the labels of the path going from the root to node **n**. Node **n** has to be labelled by a set **S** such that $S \cap H(n) = \emptyset$. If no such set can be found, the node is labelled by **@**. The idea is that every path finishing at a node labelled by **@** is a hitting set, since it intersects all possible labels for the nodes. The algorithm tries to generate as few new node labels as possible. This is due to the fact that for the diagnosis, the collection of sets **F**, which can be used as a node labels, will be given only implicitly. Calculating one element of **F** involves a call to a theorem prover to find a conflict set. The corrected Reiter's algorithm that expands the graph breadth first is:

1. *Choose one set of* **F** *to label the root node (level* **0***)*
2. *For each node* **n** *at level* **i** *do:*
    a. *If* **n** *is labelled by a set* **S***, then for every* $s \in S$ *create an arc departing from* **n** *with label* **s**.
    b. *Set* **H(n)** *to be the set of arc labels on the path from the root to node* **n**.
    c. *If there is some node* **n'** *such that* $H(n') = H(n) \cup \{s\}$, *then let the* **s-arc** *of* **n** *point to* **n'**.
    d. *Else, if there is a node* **n'** *labelled by* **@** *such that* $H(n') \subset (H(n) \cup \{s\})$ *then close the* **s-arc**.
    e. *Else , if there is some node* **n'** *labelled by* **S'** *such that* $S' \cap (H(n) \cup \{s\}) = \emptyset$, *then let the* **s-arc** *of* **n** *point to a new node labelled by* **S'**.
    f. *Otherwise, let the* **s-arc** *point to a new node* **m** *and let* **m** *be labelled by the first element* **S'** *of* **F** *such that* $S' \cap H(m) = \emptyset$ *If no such set exists, then label* **m** *by* **@**
    g. *If there is some node* **n'** *labelled by a set* **S*** *such that* $S' \subset S^*$ *then relabel node* **n'** *by* **S'** *and remove all arcs departing from* **n'** *which were labelled by elements of* $S^* \backslash S'$.
3. *Repeat step* **2** *for level* **i+1**

Theorem [Reiter87]: Let **F** be a collection of sets and let **D** be a graph returned by the algorithm above. The set **{H(n) / n** is a node of **D** labelled by **@}** is the collection of minimal hitting sets for **F**.

4.2 Hierarchical Diagnosis

In [CMW00] is proposed a very interesting application of Reiter's algorithm to find possible fault components for program debugging. The set of components and its connections correspond to parts of a possibly faulty program and the minimal hitting sets are used to guide a sequence of questions (probes) to the programmer beliefs about his program behavior. Since we propose to use this approach to debugging a student program it is necessary to have a higher level of interaction than in [CMW00]. This is done by modelling Programming Patterns as new components. Thus, the diagnosis module would be able to reason about patterns in a hierarchical fashion, i.e., to detect program faults in different abstraction levels of the program. Besides, the tutor can establish a dialogue with the student in terms of problem solving strategies as it was done in PROUST [John90], but now, based on pedagogical patterns documentation.

Figure 3 shows an example of a component model for the following problem (*Average Problem*): *Read numbers, taking their sum until the number 99999 is seen. Report the average. Do not include the value 99999 in the average*.

Note in Figure 3 that, by identifying patterns in the program model, we can construct a new model with a reduced number of components. In this approach, besides getting a model that can improve efficiency on the diagnosis process, the student will be asked to make predictions in terms of high-level strategies and goals.

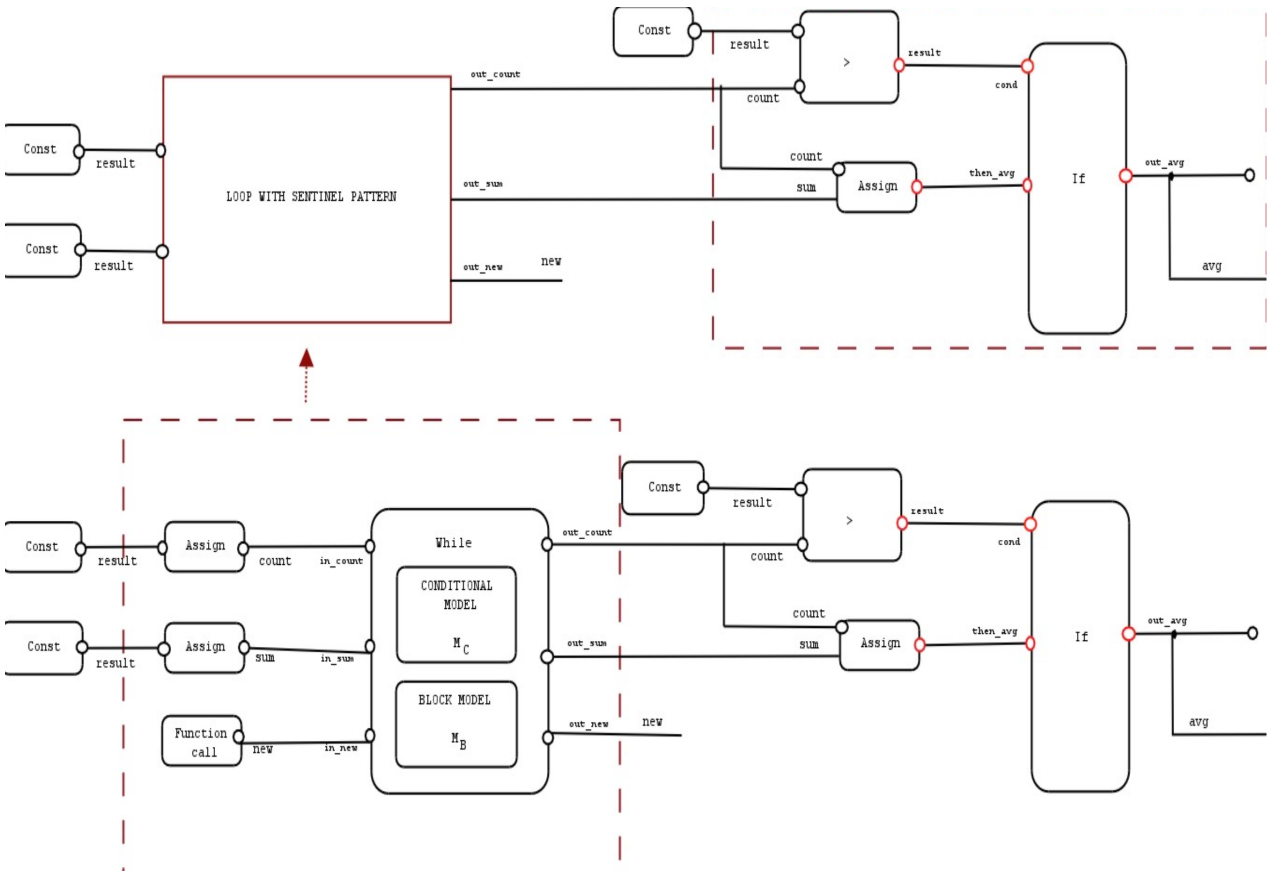


Figure 3: Two structural models of a program solution for the *Average Problem*. By identifying a pattern component in the value based model, we can make diagnosis using a smaller program description that allows for a higher level interaction with the student and a more efficient fault detection.

We have implemented a diagnosis method, based on Reiter's algorithm [Reiter87], to make the diagnosis of C programs without considering the patterns, like it was done by Mateis and Wotawa [CMW00] for diagnosing Java programs. However, we found that the interaction with the student, done basically by asking the expected value for a variable in a specific point of the program, is not enough to make the student to understand her mistakes on selecting a solution strategy. We are currently implementing our proposed new diagnosis approach where, by identifying the applied student's strategies (i.e., her selected patterns in the ProPAT IDE), we will be able to implement a better tutor interaction, in a higher abstraction level that will enable the student to identify her strategies mistakes.

5 Identifying the Student Intentions or Pattern Selection

The identification of the programming patterns used by the student can be done in the ProPAT IDE, in three different programming modes:

MODE 1. *High Control*: in this mode the student can program only by inserting programming patterns. Therefore, for each programming problem, from the exercises list of ProPAT, the teacher has to specify all the subgoals the student must program for and the student has to necessarily select a pattern to solve each one of them;

MODE 2. *Medium Control*: in this mode the student can, besides selecting and inserting patterns from the library, freely type his own code. She also needs to indicate which set of lines (probably patterns) achieves each subgoal of the problem;

MODE 3. *Free Programming*: in which the student can type his own code or choose patterns without specifying them. In this mode the tutor system should be able to automatically recognize the patterns she has used in the program.

The ProPAT tutor is currently implemented to identify the student strategy in MODE 1, only. However, this is a simplification that will allow us to focus first on the diagnosis process using patterns and the improvement of the tutor high level interaction.

6 A Practical Reasoning Model

The pattern recognition process proposed by the Pedagogical Patterns community corresponds to the ideas of rational behavior of intelligent agents proposed by the AI community. An example of such model is the BDI architecture proposed in [MEBP88]. In this architecture, a rational agent must *plan* for his goals, i.e., to reason in terms of means-end analysis and weigh the competing alternatives. The way an agent plans is based on former experiences that can be related and adapted to current problems.

In the BDI model, *Beliefs* stand for the agent's knowledge about world, including world properties and his former experiences on problem solving (plans or strategies). *Desires* are the problem goals the agent wants to solve and *Intentions* are the plans (or strategies) the agent has actually adopted to solve her goals.

The data structures specified in the BDI architecture proposed in [MEBP88] are: a plan library and explicit representations of beliefs, desires, and intentions. Additionally, there are five processes: (1) *means-end reasoner*, for determining which plans might be used to achieve the agent's intentions; (2) *belief reasoner*, for reasoning about the agent's beliefs; (3) *opportunity analyzer*, which monitors the world in order to determine further options for the agent; (4) *filtering process*, responsible for determining the subset of the agent's plans that have the property of being consistent with her current intentions and (5) *deliberation process* to produce intentions that will be finally incorporated into the agent's final plan.

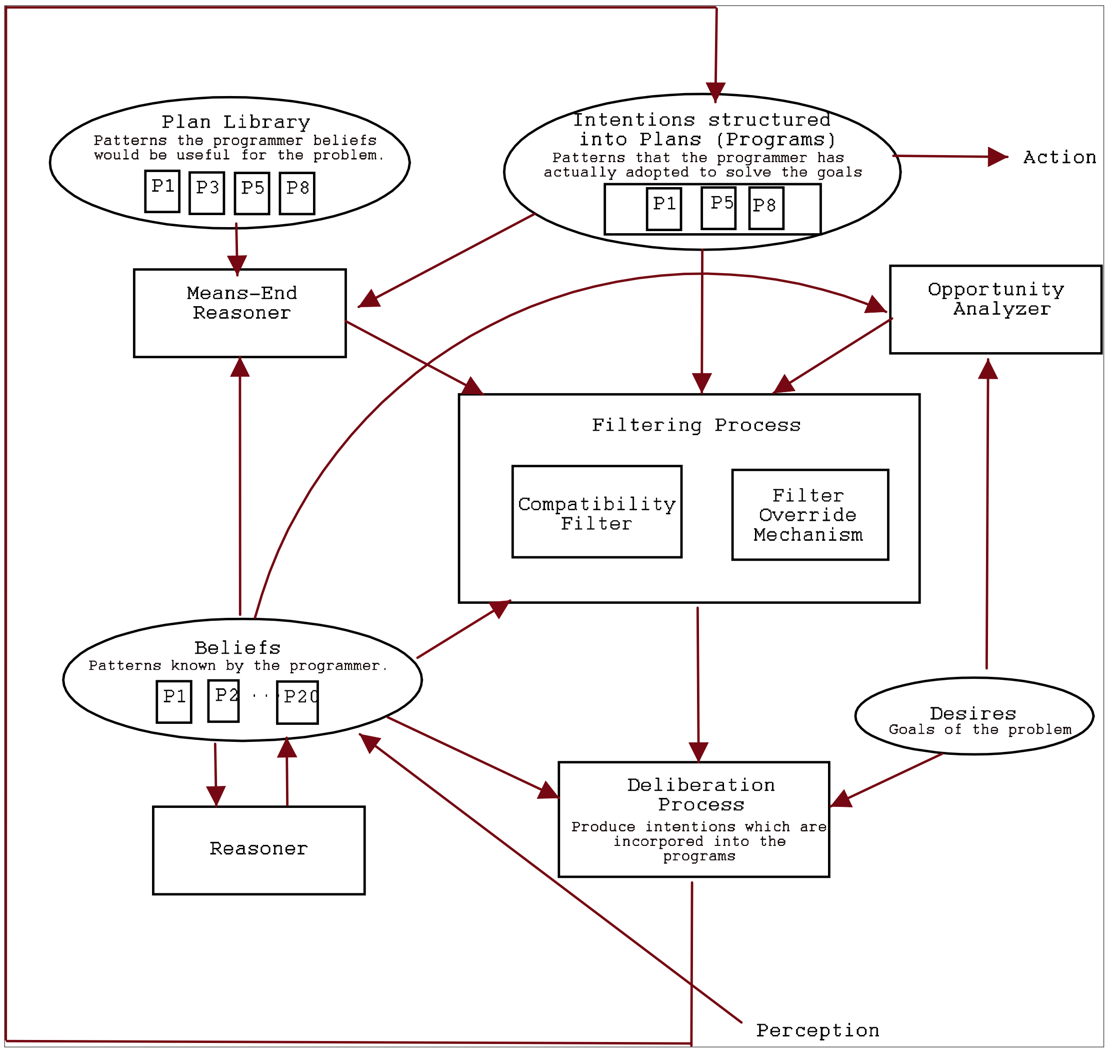
This architecture states that the intentions structured into plans constrain the amount of further practical reasoning the agent must do. The plans, as input to the means-end reasoner, they provide a clear, concrete purpose reasoning and, as input to the filtering process, they narrow the scope of deliberation to a limited set of options.

6.1 The Programming Agent

Based on the BDI architecture one can model an experienced human programmer as a practical reasoning agent. We claim that by having a theory of human rationality for programming, we can have a better understanding of the reasoning process that a programming apprentice has to learn.

Since the original BDI model considers agents that act and perceive the world during its reasoning, to make a fair comparison with a programmer, we will think about programming using an interpreter, such as a LISP interpreter. In this situation the actions are the code lines that have been given for evaluation and the perception is what is returned by the interpreter. The solution plan would be the whole code entered by the programmer. Programming former experiences can be seen as **programming patterns**. While an experienced programmer has already a full library of reusable patterns (in his mind), a programming apprentice does not have any. However, adopting the Pedagogical Pattern teaching strategy, a programming apprentice can have access to an **elementary programming patterns** library (explicitly documented) that is given by the teacher.

Figure 4 shows the architecture data flow proposed in [MEBP88] that we have modified for a programmer agent as we will explain next.



Plan Library
Patterns the programmer beliefs would be useful for the problem.
P1
P3
P5
P8
Intentions structured into Plans (Programs)
Patterns that the programmer has actually adopted to solve the goals
P1
P5
P8
Action
Means-End Reasoner
Opportunity Analyzer
Filtering Process
Compatibility Filter
Filter Override Mechanism
Beliefs
Patterns known by the programmer.
P1
P2
P20
Desires
Goals of the problem
Reasoner
Deliberation Process
Produce intentions which are incorpored into the programs
Perception

Figure 4: An architecture for a novice student agent

- **Beliefs**. Represent the student knowledge about the world described in terms of (1) the programming patterns she has learned so far, the situation they can be applied and programming goals they can achieve; (2) the value of variables in different points of the program and (3) the program subgoals that she beliefs that are already satisfied.
- **Desires**. The desires will correspond to programming subgoals of a problem description, identified by an experienced programmer (or formally specified by the teacher for the programming student).
- **Intentions Structured into Plans**. They are the programming patterns chosen to solve particular problem subgoals, i.e., programming patterns that the agent has actually selected.
- **Plan Library**. It is a subset of Beliefs and correspond to the programming patterns that the student believes they would be useful to solve the current problem.
- **Means-end Reasoner**. It is the reasoning process responsible for the instantiation and decomposition of the patterns, according to the agent beliefs.
- **Opportunity Analyzer**. It is the process that proposes options in response to (i) perceived changes in the environment and (ii) its desires (subgoals that are no longer satisfied or subgoals satisfied by previous actions). A student should be able to perceive opportunities on the use of a pattern by matching the situation she is currently in, with the pattern application documentation.
- **Compatibility Filter**. Once options have been considered, either by the means-end reasoner or by the opportunity analyzer, they are subject to the filtering process. Options that are inconsistent with the agent's existing plans and beliefs will be filtered out.
- **Deliberation Process**. Produces intentions that are incorporated into the agent's plans, for example, the student has chosen to initialize the sentinel variable with an attribution to fulfill the current intention of using sentinel loop.

Based on the assumption that a programming apprentice will have to learn how to reason as a rational agent, i.e., a BDI agent, there are many ways in which the ProPAT programming environment can give some support to the student learning process, such as:

- leaving an explicit library of patterns so the student can have access to the pattern documentation while programming, to identify opportunities to select and add patterns into her program;
- showing the programming subgoals, specified by a teacher, for a list of problems using a language close to the patterns descriptions of goals and application situations;
- allowing the student, through the Model Based Diagnosis (in a non interpreter programming mode), to express his beliefs about his program that, when compared with the execution of his program by the diagnoser, it will force her to reason about her beliefs (which corresponds to the belief revision reasoner). The idea of performing diagnosis on different level of abstraction (using patterns as components), can give to the student the opportunity to reason about a high level description of his plan/program.

Next, we want to explore, in the ProPAT, new ideas about filtering processes. These processes, in pattern based programming education, would correspond to the student capability to select the correct pattern for his programming goals, from a list of applicable patterns options and also considering the opportunities to satisfy new subgoals.

7 Conclusions

In this work, we have presented a programming environment, called ProPAT that allows the student to program using pedagogical patterns, i.e., a set of programming patterns recommended by computer educators. By using a model based diagnosis approach for detecting the student errors, we add to ProPAT the state of the art on program diagnosis and proposed a way to extend it by identifying the patterns used by the student during programming and creating a program model that includes patterns as components. This idea will allow for a better communication between the tutor system and the student.

The ProPAT programming interface is already implemented, as an Eclipse plug-in, in the programming mode *high control*. For the *high control* mode we have generated program models including some simple patterns. One of the challenges of novices' program diagnosis is the interaction with the student which can be harder than the interaction with an experienced programmer. We believe that, by using programming patterns we will be able to improve the communication. We are currently working on an introductory computer science course that will use the ProPAT IDE in classroom for testing our new pattern-based diagnosis method.

We also have shown how the BDI model, proposed by the AI studies on practical reasoning, can fit in the pattern based model of programming education, and pointed out some ways we can use this model to investigate new improvements in programming education and in the ProPAT programming environment.

Acknowledgements

This work was supported by CAPES and Eclipse IBM/IME Grant (2004).

References

[AW98] Astrachan, O. and Wallingford, E. (1998). Loop patterns. http://www.cs.duke.edu/~ola/patterns/plopd/loops.html.
[Ben93] Benjamins, R. (1993). Problem Solving Methods for Diagnosis. PhD thesis, University of Amsterdam.
[Ber99] Bergin, J. (1999). Patterns for Selection Version 4. http://csis.pace.edu/~bergin/patterns/Patternsv4.html.
[Bri02] Bridgeman, S. (2002). Intro to Computing I. http://cs.colgate.edu/faculty/stina/courses/cosc/101/f02/syllabus.html.
[CMW00] Stumptner, M., Mateis, C. and Wotawa, F. (2000). A value-based diagnosis model for Java programs. In Eleventh International Workshop on Principles of Diagnosis (DX). http://www.dbai.tuwien.ac.at/staff/wotawa/dx2000c.ps.gz.
[Greiner89] Greiner, R., Smith, B. A., and Wilkerson, R. W. (1989). A correction to the algorithm in Reiter theory of diagnosis. Artificial Intelligence, 41(1):79-88.
[John90] Johnson, W. L. (1990). Understanding and Debugging Novice Programs. Artificial Intelligence, 42 (1): 51 – 97.
[JS84] Johnson, W. L. and Soloway, E. (1984). Proust: Knowledge-based program understanding. In Proceedings of the 7th international conference on Software engineering, Florida, United States, pages 369 – 380.
[MEBP88] Bratman, M. E., Israel, D. J. and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. Computational Intelligence, 4(4):349–355.
[PC03] Porter, R. and Calder, P (2003). A pattern-based problem-solving process for novice programmers. In Proceedings of the fifth Australasian conference on Computing education, pages 231–238. Australian Computer Society, Inc..
[Pro00] Proulx, V. K. (2000). Programming patterns and design patterns in the introductory computer science course. In Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, pages 80–84. ACM Press.
[Reiter87] Reiter, R. (1987). A theory of diagnosis from first principles. Artificial Intelligence, 32(1):57-95.
[Wal01] Wallingford, E. (2001). The Elementary Patterns Home Page. http://www.cs.uni.edu/~wallingf/patterns/elementary/.