AN EXPERT SYSTEM IN OBJECT ORIENTED PROGRAMMING AREA

A.Esposito, A.Gisolfi
Dipartimento di Informatica ed Applicazioni
Universita' di Salerno
Italy

ABSTRACT.
Performing symbolic derivations is a complex problem whose solution can much benefit from the Object-Oriented Programming (OOP) paradigm. In fact, OOP nowadays is one of the most popular among the emerging software technologies since it allows modeling systems in terms that match the human thinking and language. In this paper the architecture of the expert module of an Intelligent Tutoring System (ITS) for symbolic derivation is illustrated in some details and the specific role played by OOP is appropriately emphasized.

INTRODUCTION.
The earliest tutorial systems were carried out when computers began to be seen as possible interaction tools. And it was on the basis of such a prospect that we got to the realization of CAI's systems trying to control and manage learning. The earliest CAI's on the market were very elementary; they usually were electronic books, in which the theoretic part was accompanied with a small quantity of standard exercises for each user. The system just displayed congratulations when an exercise was successfully solved, and new exercises were proposed about the topic when a mistake was made. This was one of the approaches, in which a hint supplied to the user by the system, was given as a collateral effect of exercise solving activity. A second approach could be to supply directly information relating to the subjects the user was weak about, through new theoretical texts, which could make them clearer. Till now, we just outlined the role a CAI system should play in the ambit oftutorial activity. Now, before examining the limits of a CAI system, let's see what difficulties we find when building one, or an ICAI system, that is, an intelligent system which can go by such limits.
Such difficulties are one of the reasons why such systems are scarcely found on the market. The main difficulty we find when building such systems is the same as we find when building expert systems; in fact, a Tutor is an expert system enlarged with tutorial knowledge. This difficulty is the man-machine natural language interface. In fact, the stiffness of dialogue, is owed to the fact that its rules are "a priori" fixed, and are not managed by knowledge obtained by studying the user. This is one blame on CAI's, and another one is the fact that they don't increase student's skills.
At the beginning, in favour of CAI's, it was said that one advantage they brought was the fact that their theoretic part was accompanied with exercises, which should have been carried out by students. Now, as an exercise can be generally carried out in different ways, and the system can't be open to all such hypothetically infinite ways, closed answer had to be chosen, that is, a series of options are proposed to the student as solutions of an exercise, and he has to choose the right one among them. Obviously, because of the reasons mentioned before, it's easy to understand that a CAI system in the learning ambit is severely limited.
The fact that students are heterogeneous, while CAI's behaviour is static, brought researchers towards the choice of ICAI systems. By this choice, they tried to fill the old gaps, taking care above all of tutorial activity, and availing themselves with fit pedagogical techniques for increasing student's interest in the subject.

ARCHITECTURE OF A TUTORING SYSTEM.
A tutorial system is open to knowledge, that is, it is a system learning from experience. All such characteristics make a tutorial system winning.
Till now, working and importance of a Tutor have been put into evidence in theoretical terms. Now, we will briefly show them in practical terms, putting into evidence the modules making up the system, and then, we will show the details of one of such modules, in the ambit of an implementation concerning symbolic derivation.
The architecture of a Tutor is mainly oriented to those objects representing the various knowledge elements. In it, we glossy distinguish the main components supplying the functionalities that are necessary to activate the standard tasks of a Tutor. They are:
1) Diagnoser;
2) Student model;
3) Task selector.
The "diagnoser" is a module managing man-machine interface; it can intervene without disturbing the user, and tries manage his answer.
The "student model", is a module containing information about the set of user's answers, after a tutorial session.
The "task selector" manages the set of actions the Tutor can engage in; for instance, it creates the set of exercises concerning the subjects a user has shown to be lacking about.
When working about a tutorial system, researchers must turn to two directions: on one side, from a structural point of view, creating a general framework; on the other, from a practical point of view, implementing it by an OOP language. It will be a task of

next section, to put into evidence the reasons which brought researchers to turn to OOP rather then to structured programming. We won't take into consideration OOP concepts, and deal only with the reasons which brought us to such a choice.

WHY OOP FOR IMPLEMENTING A TUTOR.

First of all, we must say that a language must possess the following properties, to be defined an OO language:
1) Data abstraction;
2) Incapsulation;
3) Binding and polymorphism;
4) Heredity.

In this section, we try to show the meaning each of these properties undertakes when an ITS (Intelligent Tutoring System), is designed and the advantages they involve as regards to traditional programming.

1) An OO language defines an abstract object, that is, a prototype of the object, describing it as a series of states the object can take on (instance variables) and a set of actions it can perform or undergo (methods). This lets us define an object as a class (a component of the Tutor), and the set of actions describing its behaviour as methods in that component.

Then, for instance, we can define the "diagnoser" object, and, for this object, a set of instance variables determining its state (for instance: "active" or "non-active") and a set of methods making that object behave, on the whole, as a diagnosis generator. That can be easily done for every component of the Tutor.

After all, as the Tutor is a highly modular system, we think it canbe useful to choose an O.O. language, as it assures such a modularity. Therefore, the designer must not worry about this point of view, as this is an inner property of the language.

2) Every object defined in an OO language has a protected memory area at its disposal, which can be accessed only by methods belonging to the class corresponding to the object. The state of an object can't be modified by actions of other objects. If we define the "diagnoser" object,it will have a reserved memory area on which the methods describing its behaviour, will work. All the problems concerning memory areas protection are a task of the compiler.

The designer, then, can assure a high modularity without great difficulties, taking advantage of properties 1) and 2).

3) We know the difference between dynamic and static binding. The work of a compiler in dynamic binding is slowed down by search of the class, the receiver of the message belongs to. Anyhow, it was empirically shown that, on the average, big software modules execution time is lesser than that of equivalent modules written in a traditional language.

Execution speed is an unnegligible aspect in the design of every module of an ITS.

A diagnoser which should interactively get a diagnosis of a user's answer,should do that in a very short time,to avoid him to feel bored while waiting; in fact, that could lead him to leave the teminal.

Polymorphism allows us to define a very lesser number of code lines than we should need with a traditional language, for getting the same results. And now we are going to explain why.

Polymorphism, a consequence of dynamic binding, allows us to get different answers according to the receivers, for fixed messages. For instance, in a traditional language, if we want to print an array containing various data types, we must create specific acknowledgement and print procedures, for every data type in the array.

In an OO language, if we have an array containing different objects, and we want to print it, we need a printing message to send to the "array" object. The message is sent by the compiler to every object in the array, and every object, then, performs the corresponding printing action; that's the reason of lesser number of code lines.

4) In OOP, perhaps this is the most important property, if connected to ITS designing. In fact, it's possible to create a class hierarchy and then define subclass relations among them. If A is a subclass of B, then it inherits from B all methods and instance variables. In a Tutor, modularity is important, but interaction between modules, through knowledge blocks sharing, is, too. We can imagine a class hierarchy providing modules with submodules, as particularizations of parent modules. For instance, we can define , as we did before, a diagnoses generator, the "diagnoser", and define "diagnoser-A" and "diagnoser-B" classes, as its subclasses.

An instance of "diagnoser-A" class inherits all methods of "diagnoser" class, so it behaves like, and it is, a diagnoses generator. Anyway, the specific methods in "diagnoser-A" class, for instance,will make so, that a type A diagnoses generator attends, to users' answer diagnosis, and not to what an instance of "diagnoser-B" class attends to.

There are no limits on the number of subclasses for a class; so, it's possible to imagine infinite kinds of diagnoses generators. The same goes for every kind of module. Heredity allows us to create such submodule specializations without repeating code lines, and with very high flexibility. On the contrary, when using a traditional language, a user has two choices:

a) to create a software module for each kind of diagnosis we are going to perform;

b) to create a software module with conditional tests calling subroutines implementing the various kinds of diagnoses.

Solution a) warrants flexibility. In fact, if we have to modify diagnosis technique A, we must only modify the subroutine implementing it; if we want to insert a new kind of diagnosis, it's enough to design a new procedure implementing it. But, it will surely happen that a lot of code lines will be repeated, and so flexibility will be obtained at the cost of waste of memory.

In solution b), we will show, in main procedure, common code lines, and in subroutines, code lines making the difference among the various kinds of diagnoses. Yet, this software module will not be flexible.

In OOP we keep highest flexibility and least number of code lines, naturally repeated without difficulties on the part of the user. Heredity has a big importance in knowledge representation, too.

We can think of putting knowledge at the lowest level, in the class herarchy we are designing. Modules interaction, that is, knowledge sharing among modules, happens by heredity.

We can think of structuring "knowledge" object in various classes, that is, according to a class hierarchy. Upper level classes represent abstract concepts, while we can show the representation of various aspects of

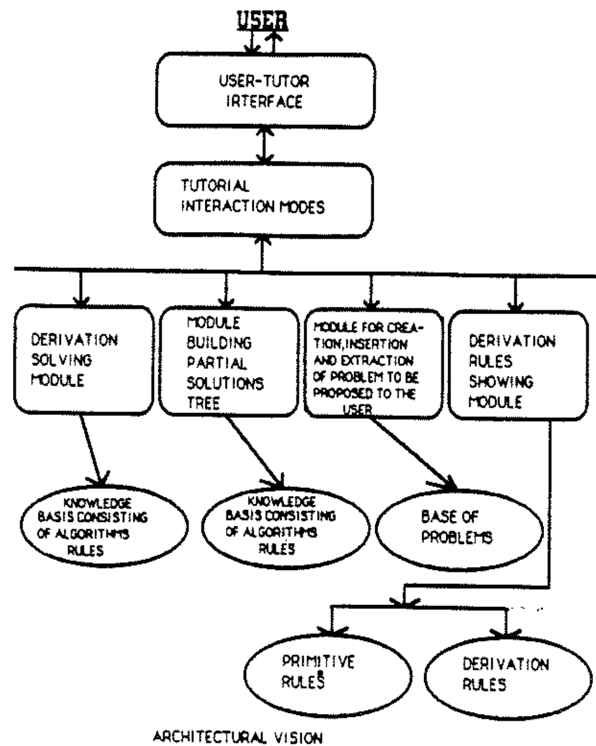
the same object, in their subclasses. Knowledge organization according to different levels of specification allows us to use only the strictly necessary knowledge, when an action must be performed on an object.
For instance, let's suppose we have an object containing pages, and have to perform the action of counting its pages. It's clear that we are not interested in classifying that object as a copy-book, a book, or a magazine. So, all we need for our action, is knowing that the object HAS PAGES.

AN O.O. TUTOR FOR SYMBOLIC DERIVATION.

Before dealing with the reasons why OOP was chosen as a fit programming methodology for implementing our tutorial system, we just mentioned which the fundamental parts of an ITS are. Now, we are going to deal with only one of the components of a specific tutor. We are going to see in detail the task selector of a tutor for symbolic derivation.
A symbolic derivation problem is defined as the transformation of an algebraic expression into another, with the property that the latter be the first derivative of the former.
As derivation problem is well determined, once derivation rules for primitive functions, and theorems about sum, product, and ratio of functions, are known, this task isn't a difficult task for an expert system. Carrying out a Tutor about this problem is more complex, as we must create an interactive environment, fit for acquisition of complete knowledge by the user.
Let's start by giving a global description of the Tutor, from a structural point of view, analysing its modules, and putting into evidence the problems connected to every single module.
Looking at figure 1, let's start with module (1): USER-TUTOR INTERFACE. It appears as a simple module:
1.1) stimulating in the dialogue;
1.2) giving a graphic view of interaction.
As to 1.1, we thought of using Socratic methodology, in which the Tutor answers the user by a question, for keeping his concentration high, and stimulating him to reasoning.
As to 1.2, we availed ourselves with Smalltalk's power in the ambit of graphic resolution. In fact, as Smalltalk supplies the user with "windows", we thought of presenting information inside of them, to make dialogue more elegant from an aesthetic point of view, and produce a more "friendly" effect than a display.
Let's go down the tree in the figure, and we find (2) TUTORIAL INTERACTION MODES. They are the following:
2.1) Demonstration;
2.2) Experimentation;
2.3) Help and advice.
In mode 2.1, the Tutor solves a problem (proposed by the user or by the system), putting the problem itself, and methods for its solution, into evidence in a window.
Mode 2.2 is essentially a high flexibility interaction mode, in the sense that the user can expose a partial or total solution, or maintain that he can't go on with the exercise. When there are solutions, the Tutor checks their correctness, and then, like a human teacher would do, shows new exercises, if the solution is total, or calls the user to continuation, if it's partial.
A different case is when a user maintains that he can't solve an exercise; this is when interaction mode 2.3 intervenes. It consists of various help levels.
The first level is used when there is a wrong partial solution: the Tutor shows a diagnosis of user's answer. The second is used when the user asks the Tutor for a first partial solution, to go on with the exercise. In the end, the third, in which the Tutor solves the exercise directly calling mode 2.1.
Now, let's go into the heart of our tutorial system, analysing (3) the DERIVATION SOLVING MODULE.
This module can be seen as a black box receiving the function to be derivated as an input, and giving its first derivative as an output. Derivation process consists of three fundamental operations:
3.1) Derivation;
3.2) Simplification;
3.3) Emission.
Let's start by saying that the user must supply the system with a function in a known form, that is, in infix notation, and must fix operators priorities by parenthesizing the expression. After that, the operations we've just mentioned are started.
3.1 is just a little expert system, supplied with an inferential engine whose activity is performed on a knowledge basis consisting of algorithmic rules.
The first operation it performs is the transformation of the infix expression into prefix form. We need that as for a fast recognition of the expression, as for the following operation the inferential engine is going to perform: extraction from the new expression of unary and binary operators.
Probably, this operation is owed to the fact that unary operators take to elementary derivation, while binary operators need application of known theorems; therefore, we can have the great advantage of knowing in advance whether derivating elementarly, or by theorems application.
After this preliminary operation, the inferential engine performs derivation through algorithmic rules we are not going to deal with, now, for the sake of brevity.
3.2 is a little expert system, too, fit for algebraic simplification on the prefix form symbol set it is supplied by 3.1. The last solving submodule we analyse is 3.3. It is an expert system, again, working in two stages. The first is said "finishing touch" and the second "presentation" to the user of the result. Both of them are a further simplification work, carrying out the opposite step of 3.1, that is, transformation from prefix form into infix, so that the user can get a clear and readable result.
Now analyzing (4) the MODULE BUILDING PARTIAL SOLUTIONS TREE.
In the ambit of such module, we must take into consideration two problems. The first is "how to define partial solutions", and the second is "how to manage derivation modalities".
While the first is not a big problem, if we put the expression part that has not been derivated yet, as a derivation function (for instance, given the problem SIN(X) + COS(X), we get the partial derivative COS(X) + D[COS(X)]). The second grows large when for every operator we have several derivation ways. For instance, we have five following forms for operator "+":
D[ARG1 + ARG2] ; D[ARG1] + D[ARG2]; D[ARG1] + (ARG2)' ;
(ARG1)' + D[ARG2]; (ARG1)' + (ARG2)' .
The problem is partially solved if we correctly build the partial solution tree, in which every child node of

the root (representing the starting expression) represents one of the possible derivation ways for the operators in the expression. The module building such a tree is an expert system, in which the inferential engine performs its activity on a knowledge basis consisting of the various application modalities of the derivation rules.
The tree is carried out by imagining every node as a frame.
Slots initialization in the ambit of such frames, consists of memorizing partial solutions as in infix form, as in prefix form, after the transformation. This is repeated at every step of partial derivation, expanding the tree containing the frames of all the possible partial derivation ways. Therefore, this implementation technique is practically the work of the diagnoser when analysing a user's answer; in fact, as the knowledge basis the inferential engine works on, is complete, a wrong answer by the user is just a non-existent branch in the tree, when going from level "i" to level "i+1" of derivation operation.
The module (5) is the MODULE FOR CREATION,INSERTION AND EXTRACTION
OF PROBLEMS TO BE PROPOSED TO THE USER.
This module's task is to generate problems the Tutor will supply the user with, as educational material.
In OO language implementation, such a module consists in creating a frame in correspondence with a fixed derivation rule, for instance, when a user has shown lacks about it.
Such a frame consists of two slots. The first contains the derivation rule, and the second, the association dictionary. By association, we mean a couple (key,value) in which the key represents the difficulty level of problems concerning the rule contained in first slot, and the value, a circular queue of problems, where the key represent priority to be assigned to the queue.
As to 5.2, problem insertion, the main difficulty lies in building the second slot we mentioned before, that is, to compute the difficulty level of the problem through fixed parameters. The integer so obtained is the key to which the circular queue where the problem is to be inserted, will be associated.
5.3, extraction method, is almost like insertion, with the difference that the extracted problem (already shown to the user) is inserted into the queue again. The reason of re-insertion is to create a turn-over on the problems to be proposed to the user with a fixed difficulty level.
The last module (6) is the DERIVATION RULES SHOWING MODULE.
This module is the last we take into consideration. Of course, in the ambit of derivation operation, as we mentioned at the beginning, we must make a distinction between primitive rules, and rules springing from theorems. Showing a rule is a help level proposed by the Tutor in "windows", which a user can call when he doesn't remember how he has to apply a rule.

CONCLUSIONS.
At the end of this work, you might legitimately wonder what you've found in it. Sure, you did not learn how to become an ITS designer, what we can't teach you. But, on the other side, our aim was to transfer to the reader, our interest in the study of computer science as a support of education, with its numberless problems, and its evident powers, too.



ARCHITECTURAL VISION

REFERENCES.

[1]J.R.Anderson, B.C.Franklin, B.J.,Reiser, Intelligent Tutoring Systems, Science, Nr.2, vol228,(1986) pp.456-462.
[2]R.R.BurtonJ.S.,Brown, An investigation of computer coaching for informal learning activities. Int. Journal on Man-Machine Studies, vol.11.(1979).
[3]R.Kimball R. A self-improving tutor for symbolic integration. In Intelligent Tutoring Systems. Ed. by D.H.Sleemann,J.S.Brown. Academic Press Inc. pp.283-307.(1982)
[4] D.Sleeman,J.S.Brown, Intelligent Tutoring Systems. Academic Press Inc. pp.1-10.
[5]J.nar,R.nningham,J.hultz An Object Oriented Architecture for Intelligent Tutoring Systems. OOPSLA '86 Proc. pp.269-276.(1986)
[6]M.R.Genereseth M.R. The role of plans in ITS's. In Intelligent Tutoring Systems. ed. by D.H.Sleeman, J.S.Brown, Academic Press Inc. pp.137-155.(1982)