Towards the Generation of Tutorial Courses for Applications

García, F.

Escuela Técnica Superior de Ingeniería Informática
Universidad Autónoma de Madrid (UAM)
Cantoblanco, 28049
Madrid, Spain
Federico.Garcia@ii.uam.es

ABSTRACT In this paper we describe the underlying ideas of a system for the development of tutorial courses for interactive applications. This environment makes use of the user interface design technology based on declarative interface models. In particular, it uses hierarchical user task models to provide the designers with the possibility of generating tutorial courses about how to use highly interactive applications. The aim is both to reduce the generation and maintenance costs for the design of these tutorial courses and to build more dynamic and powerful tutorial course systems. The environment we propose is based on two separated modules. The first one, Advanced-HATS, is aimed to deal with the interactive teaching of isolated user tasks. In order to teach users how to perform a particular user task, this module uses the information in the user task model and the information provided by the underlying user task management system at run-time. The second module, the *scenario execution module*, is in charge of preparing the adequate *scenarios* for the first module to teach the users. Finally, CACTUS, a prototype tool implementing these ideas, is introduced.

1. Introduction

Current applications enclose a complexity that was unthinkable not many years ago. These systems, whose complexity is a consequence of the capabilities offered by the modern graphic user interfaces, have in many cases hundreds of commands, often operating in a different way depending on the context. This causes that only very advanced users can take advantage of a high percentage of the power of these systems. However, these tools rarely come together with systems that let the novice users systematically learn how to use them . This user profile is the focus of this paper.

We introduce the underlying ideas of a system aiming to palliate the problems related to current guidance components of software systems. It offers an architecture to design completely dynamic interactive tutorial courses. These courses can follow-up the activity of the pupils during their teaching sessions and act consequently depending on

the user actions. Moreover, there is a drastic reduction in the costs of designing and maintaining the tutorial courses.

The system we propose lies upon the programming technology based on declarative user interface models (Szekely 1993), that provides many benefits in the development of applications (Puerta 1998), like partial design automation, validation of the solutions, and so on. The information in these models makes it possible for application-external tools to reason about the state of the application at run-time, in order to modify the interface behavior. In our particular case, the use of these models makes it possible to give the final users, that is the pupils, the possibility of receiving feedback from the continuous following-up of their activity and the evaluation of their knowledge on the tasks taught through the tutorial course.

We propose a system based on two modules. The first one, Advanced-HATS, is a tutoring system in charge of teaching user tasks. This subsystem provides users with context dependant dynamic messages in addition to several other innovative tutoring capabilities, such as context dependant graphical feedback. The second module, the *scenario execution module*, is in charge of preparing the appropriate contexts for the teaching sessions proposed in the tutorial course.

These ideas have been implemented in CACTUS, a prototype for developing courses. This system has been tested in the generation of tutorials courses for several applications. In particular, it has been proved with interactive interfaces for continuous digital simulation of systems such as the solar system and Volterra equations (Alfonseca 1998). It has also been tested in the generation of a tutorial course for the OOPI-TasKAD application, prototype for an object-oriented computer aided design environment based on the prototype/instance paradigm. Furthermore, tests have taken place on an electronic agenda that allows the typical options of this kind of tools. Finally, our tutorial course generation tool has been tested to generate some tutorial courses for *Schoodule*, an application that helps to generate school schedules.

This paper is structured as follows. First, we will summarize the problems associated with current tutors in the software context and the most relevant research projects in this field. Next, we will introduce the architecture of the system, including ATOMS, the user task management system we base our architecture on, the Advanced-HATS tutoring system and the *scenario execution module*, using examples to illustrate their operation. After that, the CACTUS tool will be briefly described, followed by the conclusions and future research lines of this work.

2. Context and Related Work

Even today, in most of the cases, the guidance component that complex applications offer to their users is based on hypermedia explanations. These components describe how each application command can be accessed and which functions they are supposed to perform. Obviously, this format has serious drawbacks related to their contents (Contreras 1998), that limit the usefulness of that kind of help system:

- The explanations are given at a very low level. The messages are given as a set of recipes, interconnected through hyperlinks, referring only graphic interface objects, and they never refer to higher conceptual level tasks. As a consequence, the explanations are not user task-oriented, but they are widget-oriented.

- Isolation of the help component. There is no interconnection between the application and the help system. Thus, the help component can not communicate with the user by accessing the application user interface.
- The help has a static style. The processes are fully explained before the user begins to perform the task, having her/him to switch between the help window and the application one to read the explanations and perform the task in parallel.
- There is no feedback from the help system towards the user, to indicate her/him when any of the steps s/he is accomplishing is not correct according to the system indications. Thus, the help system is completely passive and unable to correct or realize whether the user has understood its explanations.

Very few applications deliver, along with them, interactive systems to guide the users in the application learning. In those cases when a tutor for an application is included, there are very important drawbacks that make its usefulness/cost proportion too low to be produced for applications that will not be of wide use:

- Huge development costs of the teaching tool. The tutors usually simulate the application and the users have the impression that they are learning the real application, not just a simulated one, and the creation of that copy usually implies very high costs.
- Distance between the real work and the work with the tutor. Current teaching tools do not usually permit the learning using the work context of each user, but they only teach by means of predefined examples.
- High maintenance costs. Usually, each change in the application should be reflected in the tutor to guarantee the soundness between the application and what the teaching tool explains.

In the last years a great effort has been devoted to generate help for interactive applications using different paradigms. Next we will describe how these projects have surpassed some of the drawbacks mentioned above.

Cartoonist (Sukaviriya 1990) is a system based on the UIDE environment (Foley 1988) of generation of user interfaces based on declarative models. This system uses animations to communicate the information to the user. Cartoonist uses backchaining to navigate through the preconditions and postconditions of the commands specified in the model. The information this system produces is not hierarchically organized, so the explanations lack of the structure provided by hierarchical descriptions in terms of goals and subgoals.

On the other hand, the work of Pangoli and釘Paternó (Pangoli 1995) was pioneer in offering help from user task models. Thus, the information the user receives has greater semantic power than the one received from current help systems. However, it does not follow-up user activity, and then it does not offer her/him feedback about the task accomplishment.

Teach me While I Work, TWIW (Contreras 1996) also produces task-oriented help, but it goes a little beyond a help system and provides teaching capacities. In this sense, not only it informs users about the steps they should follow to accomplish a task, but it also filters user actions that are not correct according to the indications provided. One of the biggest problems of this system is that the task model that it uses does not include any contextual information, and it allows very limited types of sequencing among tasks. Moreover, this system is mostly oriented to users who have some familiarity with the applications they want to learn, but it is not intended to be used by novice users.

Finally, Help for ATOMS Task System, HATS (García 1998a), is a help system that uses user task models in order to generate the explanations. It has been later endowed with similar teaching capabilities to those of TWIW, as we will see when describing the Advanced-HATS tutoring system. In particular, HATS provides the user with dynamic help, updating the help messages according to what s/he needs to perform at each moment. The most novel feature of this system is its ability to offer graphical feedback, by highlighting application graphical objects, about what has been previously made and what is still to be done. HATS is based on the task models used by ATOMS (Rodríguez 1997, García 1998b), a system that manages user task models at run-time and allows the tasks to incorporate and to manage contextual information, in the form of parameters.

3. Architecture

The general architecture of the tutorial course generation system introduced in this paper is shown in Figure **Error! Unknown switch argument.**. As we can see, our proposal takes advantage of a refinement of the HATS help system, the Advanced-HATS tutoring system, and a scenario *execution module*, to provide the course designer with an environment with which, at the lowest cost, s/he can provide users with advanced interactive courses on their applications. The tutoring is oriented to teach the tasks that can be carried out in the applications, while the environment allows the tutorial course designers to specify the previous *scenarios* adapted for the teaching of those tasks. Both modules, Advanced-HATS and the *scenario execution module*, are built on top of ATOMS in order to get task-oriented support. In the next subsection, we will describe ATOMS in some detail.

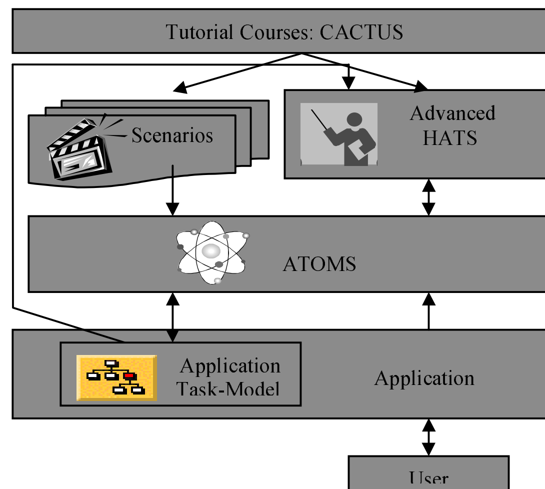

Figure Error! Unknown switch argument.**: Architecture.**

In the rest of the paper, we will illustrate the system features using examples from a tutorial course developed for *Schoodule,* an application that uses a database and a constraint solver to help on the generation of school schedules. The database can be described in a simple way by means of four entities and one relationship. The first entity corresponds to schoolteachers. The second one describes the different subjects that can be taken in the school. The information dealing with the groups of pupils is reflected in another entity, and the last one deals with the classrooms. Finally, the *assignment* relationship indicates which teacher imparts each subject for each group of pupils. From

this information, the application automatically generates a schedule proposal, indicating both the timetable and the classrooms for each lecture.

3.1. ATOMS

ATOMS, Advanced Task-Oriented Management System, is a run-time user task management system. It is not a system aimed to design application user-interfaces from scratch, as TRIDENT (Vanderdonckt 1995) or ADEPT (Johnson 1995), but it is aimed to allow other processes to request to be notified when tasks are completed and invoke application tasks. These facilities support the constructions of agents that can assist users in various ways.

ATOMS based applications have to define their user task models, hierarchical representations, using a declarative language, specifying the tasks that can be accomplished with the applications and the rules establishing the relationships between different tasks. A detailed discussion of the advantages of hierarchical task models can be found in (Zeiliger 1997).

3.1.1. ATOMS Task Models

An application task model has to define two kinds of tasks: atomic ones and composed ones. The first ones model the interactions a user can directly perform with the application interface, that is, they describe the abstract interaction object associated to each atomic task. On the other hand, the composed ones allow modeling the top-level tasks a user has in mind when s/he carries out a certain set of actions. Both of them may have associated parameters that act as contextual information.

The task models, in addition to defining the application tasks, define the rules that hierarchically relate the tasks to each other. Each rule relates a particular composed task with a certain set of atomic and composed tasks. Furthermore, each rule includes other types of information. Some of them are:

- The execution relationship between the subtasks. This relationship can be:
  - 'sequence', where a subtask can not begin until every previous subtask has finished.
  - 'and', where the execution a of each subtask is independent of the execution of any other and everyone has to be accomplished.
  - 'xor', where executing any subtask implies that the task including them finishes.
- Description of the parameter flows between tasks. This description determines how the parameter values are obtained and how they are transmitted between tasks along the hierarchy.
- Which possible preconditions and postconditions must hold before and after a subtask execution.

3.1.2. ATOMS Internal Operation.

With respect to the internal operation of ATOMS, it is composed by four main blocks (see **Error! Unknown switch argument.**): the *Dynamic Application Tasks*, the *Parsing Engine,* the *Task Modeling Tool* and the *Emulation* module. The first two modules follow-up the user activity while interacting with the application, in order to recognize at any moment which tasks s/he is carrying out. The third module is used for the

interactive generation of the ATOMS task models. The last block executes tasks using animations, and it is provided as a service for external value-added tools to modify the application interface behavior at run-time. These four blocks will be analyzed in this section.

The first block, the *Dynamic Application Tasks*, represents both the tasks the user has accomplished throughout the work session (*Historic*) and the tasks s/he is currently carrying out (*Active Tasks*). ATOMS updates this information at run-time, as the user carries out the tasks specified in the task model. The main goal of the *Dynamic Application Tasks* module is to allow external value-added tools, such as Advanced-HATS and the *scenario execution module*, to reason about the application state in order to provide users with extra services. The second block, the *Parsing Engine*, analyzes the user interactions with the application. Taking into account the application task model and the state of the current tasks (*Active Tasks*), the *Parsing Engine* determines the state of the tasks, and updates the *Dynamic Application Tasks*. The role of the *Parsing Engine* is similar to the one played by *parsers* in *Natural Language Processing*, the lexical tokens being substituted by user interactions. Similarly, the Application Task Models play a role similar to *Unification Grammars* (Maxwell 1994).

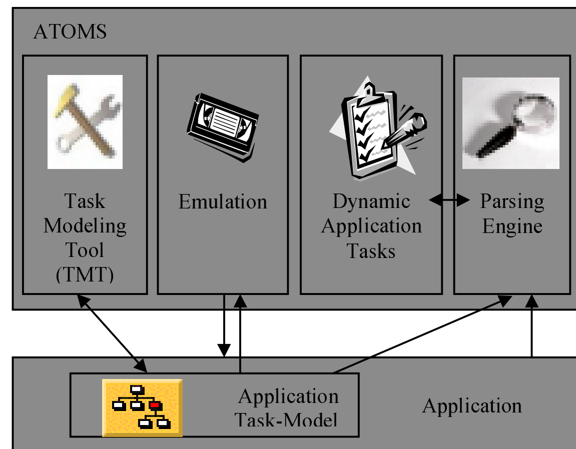

Figure Error! Unknown switch argument.**: ATOMS architecture.**

Task Modeling Tool.

The *Task Modeling Tool* (TMT) aims to palliate the greatest drawback of model based technology: the designer still has to model the application, so s/he has to get used to a new programming language. ATOMS offers TMT to allow the designer to specify the task-model once her/his interactive system has been built. This a-posteriori specification of the application user task model is a consequence of the run-time nature of ATOMS. This specification is done interactively, using both visual programming and programming by example techniques (Cypher 1993).

On one hand, TMT provides support to capture interactions and to generalize interactively them, thus allowing defining through examples the atomic tasks. The designer indicates the tool that s/he is going to perform an atomic task and s/he carries it out. Afterwards, s/he modifies it using a specialization and a generalization process. If the designer does not modify it, then ATOMS will consider that any interaction of the same type to the one provided will match. For example, if the designer has moved a

graphical object, the interaction would be described as '*moving any object*'. An interaction modification includes two steps: the interactive selection of the set of relevant attributes from the interaction and the generalization of the values for those relevant attributes. The first step identifies which values ATOMS must have into account to match a user interaction against the one the designer is accomplishing. For example, the designer can indicate that the color of the object being moved is important, but the object size is not. The model designer tells TMT which attributes are relevant by selecting them from a list of attributes of the interactor object representing the interaction. This selection is a specialization of the interaction, since TMT forces the attributes to have the same values they had in the first example of the interaction. For example, if the moved object had red color, the interaction would be described as '*moving any red object*'.

The second procedure *generalizes* the relevant attributes by allowing the designer to repeat the interaction in a similar way. The system generalizes interactions using a rather simple algorithm that looks for a maximally specific expression representing a set of interactions (Mitchell 1982). For example, if in a second demonstration the user moves a green object, then the interaction will be described as '*moving any red or green object*'. After a demonstration, the system generates an expression including the values for the relevant attributes during each demonstration. The final representation of the interaction being defined, i.e. the description of its attached abstract interaction object, is referred to as an *abstract interaction pattern*. In the current TMT implementation, *equality* and *or* predicates are generated for each relevant attribute. A pattern represents the conjunction of all the expressions corresponding to each one of the relevant attributes for a particular interaction object. With just these two operators, TMT generates *abstract interaction patterns* in which each attribute value contains a disjunctive expression of equality tests. However, as ATOMS is able to manage general predicates, such as arithmetic and logic operators, more complicated expressions can be manually specified by advanced designers in order to declare *abstract interaction pattern* for atomic tasks.

On the other hand, TMT provides support to define interactively composed tasks and rules relating the tasks. As in the case of the predicates for the *abstract interaction patterns*, TMT does not allow to generate interactively every possible rule. The only expression for parameter transfer functions that TMT interactively generates is the identity function. Although this is a very common parameter flow function, advanced designers can manually specify several other functions, since ATOMS can manage them.

Emulation.

Finally, the fourth main module in ATOMS, *Emulation*, is entrusted with executing tasks using animations. This module is provided as a service to be used by external value-added tools in order to modify the application interface behavior at run-time. As we will see later, this module is the basis of the *scenario execution module.* During the execution of these tasks, a mouse-pointer appears moving along the screen in the same way a user would make it move by using the mouse to accomplish them. In addition to this, the same interim feedback and the same application behavior are obtained as if the user had accomplished the task. This module is capable of emulating the user interactions by analyzing the tasks and rules descriptions specified in the application task model. Starting from the model, the description of the task we want to emulate and

the values of the parameters that are provided, *Emulation* decides which atomic tasks must be executed, as well as the corresponding parameter values and the order in which they must be carried out. Since each atomic task has an associated *abstract interaction pattern* that describes the corresponding interaction, *Emulation* analyzes the aggregation relationships of graphical and interaction objects in the target application in order to determine which steps to accomplish and with which values that interaction has to be performed. For example, this ATOMS module could be requested to execute the task *SelectTeacher* with the parameter *name* having the value *"John Hopkins"*, and it will execute it by means of animations, in the same way a user would accomplish it.

Emulation receives task execution requests in which the specified values for the parameter references parameter values for the top-level task, that is, parameter values of the composed task to be emulated. Since this module executes the top-level tasks by progressively decomposing them into subtasks to obtain the atomic tasks to be executed, one of the most important steps this module performs is transforming the values of the composed task parameters into simpler task parameter values. To perform this transformation, the inverses of the parameter transfer functions of the task model rules are applied, and the parameter values are propagated down along the task hierarchy. Thus, in the previous example, the textual parameter "*John Hopkins*" corresponding to the name parameter of the *SelectTeacher* task will be converted into a parameter, whose type is a graphical object and whose text content is "*John Hopkins*". This label will make *Emulation* to search for a suitable graphical object during the execution of the *SelectTeacherFromList* atomic task, thus selecting the correct item from the corresponding list.

Emulation also admits task execution requests in which one or several of its associated parameters do not have any value assigned. For example, this module accepts requests to emulate tasks like *SelectTeacher*, with no specified parameter values. *Emulation* treats this kind of situation, in which the value of some parameter that has not been provided is needed, by indicating the user that s/he must provide the parameter value in an interactive way. In the cases in which it is possible, *Emulation* also offers feedback to the user on what possible values or options are available. In the previous case, *Emulation* has not been provided the teacher's name, so it does not know which element from the list of teachers must be selected. Hence, it makes every item in the list blink, and indicates the user that s/he must select one of them. From that moment on, the system only allows the user to select one item from the list, and disables the rest of the available interactions in the application. In general, not every parameter transfer function is bijective. Thus, when non-bijective functions are used to specify parameter transfer flows, it is not possible to convert a parameter value in its corresponding lower-level parameter values. This means that there are cases in which a certain composed task parameter value is not fully promoted to an atomic task parameter value. Of course, the selection of appropriate default values may be a suitable solution for providing inverse functions, but this would imply having more knowledge about the parameter semantics, something that would include undesired complexity into our task models. However, in many cases it is not a dramatic trouble, since *Emulation* will try to execute the task, independently of whether a particular parameter value has been successfully promoted or not. If any value needed to perform an interaction is not available, then the system will ask the user to provide it interactively.

3.2. From Tasks to Tutors.

In our environment, a tutorial course is composed by pedagogical units, each one teaching a set of tasks that are related by some criterion. Before teaching a task, the appropriate context for the teaching session may be prepared by means of the execution of a scenario.

As it has been already mentioned, we propose an architecture based on two modules (**Error! Unknown switch argument.**). The first one, Advanced-HATS, is entrusted with teaching the user to accomplish a certain task. The second one, the *scenario execution module*, prepares the appropriate contexts for the teaching sessions, in order to make it possible to carry out the teaching of the tutorial course tasks.

The Advanced-HATS module is a refinement of the HATS help system (García 1998a), whose most important improvements are centered on the incorporation of tutoring capabilities to the helping capabilities already mentioned in section 2. In addition to HATS features, Advanced-HATS offers the possibility of acting with several degrees of flexibility. As a consequence, the system not only indicates the user how s/he must carry out a particular task, updating the explanations as s/he accomplishes the task, but it can also filter some wrong user actions, compelling her/him to follow a correct path. This filtering will be performed depending on which degree of flexibility is set. Another Advanced-HATS capability, not present in HATS, is the possibility of teaching a user how to perform parameterized tasks. This means that Advanced-HATS can give explanations related to the task parameter values in the messages conveyed to the user. For example, HATS is only able to teach tasks without any parameters, like '*Add a new teacher*', by indicating the user which interactions s/he has to perform and, in particular, in which text-input field s/he has to input the name. In contrast, the Advanced-HATS tutoring system is able to teach parameterized tasks like '*Add a new teacher with name John Hopkins*' by indicating that s/he has to input the *'John Hopkins'* text string in a particular text-input field. The tutoring system also filters actions that, although they are correct in order to accomplish the explained tasks, are not with respect to the values adopted by the task parameters, thus forcing the user to perform the task with the desired parameter values. In the previous example, it would filter any interaction aimed to input a name different from *'John Hopkins'*.

With respect to the *scenario execution module*, it prepares the necessary contexts to carry out the teaching of the tasks proposed in the tutorial courses. The *scenarios* are procedural descriptions, written in an specific programming language, of processes including references to application tasks, variables, conditional constructions, iterations, and so on. The *scenario execution module* interprets the corresponding procedural descriptions. For example, the unit in the *Schoodule* tutorial that teaches the user how to add a new teaching *assignment* to the application database previously prepares a scenario to add some default teachers, classrooms, and so on, to the database. In this way we can be sure that the user will not have any problem during the practice, for example if there is no teacher in the database to choose, or if there is no classroom in the database to select. The course designer may include in any unit a scenario followed by a task teaching session. Then, the execution of that unit will cause the system to ask the *scenario execution module* to prepare the *scenario* to change the state of the application. In our example this change adds some teachers, classrooms, etc. to the application database. After this, Advanced-HATS will teach the user how to add a new *assignment* to the database.

Task references in this language play the same role played by system calls in other environments: the interpreter, basing on the ATOMS' *Emulation* module, executes that

task using animations. It is also possible to build *scenarios* interactively by performing the desired tasks in the application and telling the system to generate a scenario from those tasks. For example, the designer can perform the tasks for inserting some default teachers, classrooms, groups and subjects in the application database. After that, the designer can create automatically a scenario to perform those actions.

Nevertheless the environment allows users to practice the tasks in their own work context instead of using predefined *scenarios*, in order to provide designers with the biggest flexibility. Thus, the user can choose between three different behaviors: always execute the *scenarios*, never do it, or just choose whether to execute them or not. The fact of not executing a scenario may imply the impossibility of learning a particular task, because it may be impossible to perform a particular task in the user's current context.

4. The Implementation

CACTUS is a system for the interactive development of tutorials, based on the ideas explained before. It aims for releasing the tutorial designers from most of the intensive workload of developing the tutors.

CACTUS integrates designer oriented services, such as support for creating, editing, testing and debugging of tutorials, and pupil-oriented services, such as support for the execution of the courses. A course is composed by units, each one teaching a set of user tasks related by some criterion. CACTUS allows designers to create and modify interactively the tutorials, using programming-by-demonstration techniques, similar to the ones implemented in TMT.

CACTUS uses the metaphor of representing a tutorial as if it were an interactive textbook, and associates the most important parts of the tutorial with representative parts of a textbook. Most parts of the interactive books are automatically generated by CACTUS, and the designers only have to include the desired contents for the pedagogical units. The main advantage of our courses with respect to those generated with existing systems is the fact that our courses follow-up user activity. This makes possible to provide users with context-dependant explanations. Thus, the user does not learn by reading the book contents, but s/he learns by executing them and by following the indications they provide. This is completed by offering the use of predefined *scenarios* as context for the learning process, in addition to using the user's own work context. Another important feature is the automatic incorporation of hypertext links, so that it is easy to navigate through the books while the designer is released from explicitly providing the hyperlinks. In the rest of this section we will give a description of the structure of a CACTUS book.

First, CACTUS generates a cover like the one in **Error! Unknown switch argument.** (left). It includes the course title and a general description to provide an overall idea of the course purpose. Both fields are user-customizable.

Afterwards, CACTUS automatically generates an index of the course pedagogical units using the same order the designer follows to develop them, as shown in **Error! Unknown switch argument.** (right). Each entry contains a customizable unit title and a description of its contents. For each entry, a hyperlink is generated to the starting page of the corresponding pedagogical unit. The contents of a pedagogical unit will be explained later.

One of the main aspects of a CACTUS course is that it can be followed using several orders. In this sense, our interactive books are similar to those textbooks used in advanced courses, which usually have a predefined order to be read, but can be read using other orders, since the matters covered are complementary. CACTUS courses include the sequencing notion between the different pedagogical units of the tutorial. CACTUS automatically manages the sequencing, and permits a user to execute a unit depending on whether its previous units have already been taught or they have not. The ability to follow user interaction allows the system to manage automatically the unit sequencing. The next two parts of a book are narrowly related to this sequencing notion.

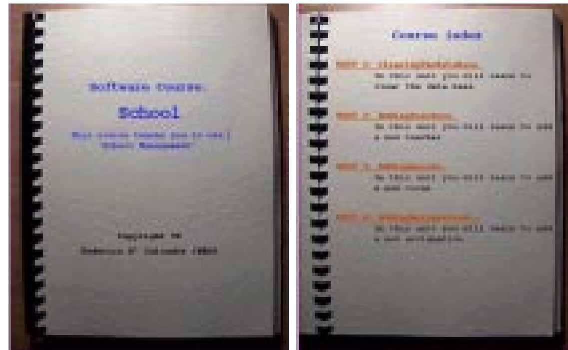

Figure Error! Unknown switch argument.**: A tutorial book cover and index.**

First, a directed graph representation of the course structure is generated, as shown in Figure 4 (left). Each node represents a pedagogical unit, and its color represents its state. The state of a unit indicates if it has been executed, if the unit has not been executed yet but it is accessible, or if the unit is currently blocked because some of its predecessor units have not been accomplished yet. An arrow from node A to node B indicates that unit A has to be performed before unit B. The states of the units are automatically updated by CACTUS as the user completes the course.

Second, a proposal of a sequencing of units for the course is generated, as shown in Figure 4 (right). In this proposal, when a node is visited all its predecessors have already been visited, so that it will never be the case that the user tries to access a blocked unit.

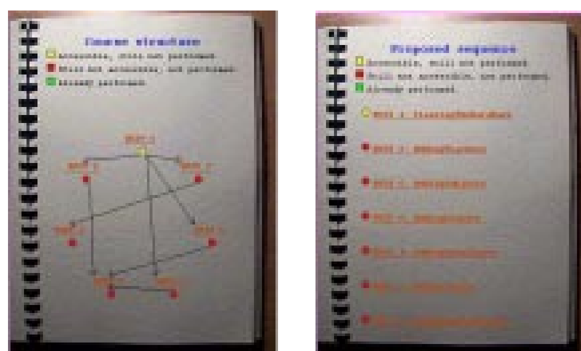

Figure Error! Unknown switch argument.**: Tutorial book graph and proposal.**

Afterwards, as many chapters as pedagogic units in the tutorial are presented (Figure 5, left). Each unit incorporates some tutoring sessions and, probably, some *scenarios* to execute. The page layout includes explanations about the tutoring sessions, the *scenarios* to be executed and images. CACTUS automatically generates explanations indicating the users the different things to be done, during the execution of that unit,

from the references of the tasks to be taught. Of course, this automatically generated explanations are fully customizable by the designers.

Finally, CACTUS automatically generates a task glossary (Figure 5, right) at the end of the book, where every task being taught in the course can be found. Each entry relates a task with the list of units containing tutoring about it, and a hyperlink allows to easily access those units. Then, the process of searching about a certain task is reduced to a look-up process and to follow a hyperlink. This glossary is transparently managed by CACTUS, so the designer does not have to worry about updating it.

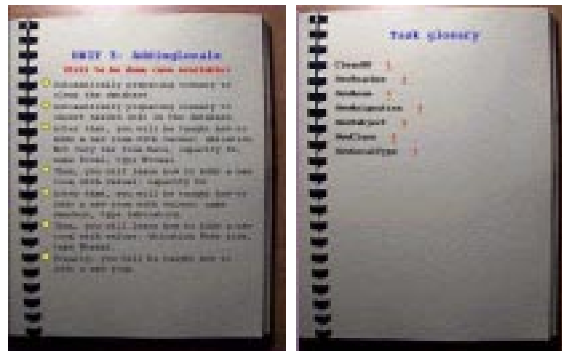

Figure Error! Unknown switch argument.**: Tutorial book unit and task glossary.**

4.1 Tutorial execution.

CACTUS allows executing some parts of the book to teach a user how to perform some user tasks using an interactive software application. The execution of tutorial courses can be also used by course designers to test and debug them at development-time.

Some sections of the book have a group of instructions associated. Instructions in this environment can be classified in three groups. The first one includes those for specifying task-teaching activities, preparing suitable *scenarios* to practice with, and showing users information about their performance. This kind of instructions has an immediate feedback on the application interface. The second ones do not have an immediate feedback towards the user, and includes low-level specifications such as variable assignments or instructions aiming to format the course (unit descriptions, unit titles, images, and so on). This group also includes control structures such as loops, conditional blocks or random execution blocks. Finally, a third group controls how the *scenario execution module* and Advanced-HATS operate, including setting the degree of flexibility for the tutoring module, the quantity of messages to be shown during the teaching activities or the *scenario* execution speed.

The executable sections of an interactive tutorial course book are the pedagogical units and the proposal section.

The pages of the pedagogical units usually contain task-teaching activities, examples of predefined tasks, *scenario* preparation instructions and feedback messages for the users. These kind of instructions are adequately related by the use of different types of control structures.

For example, in our course for *Schoodule* there is a unit that explains how to add a new teaching assignment. In this unit, the first thing the tutorial course performs is preparing a scenario. This *scenario* adds some default teachers, subjects, groups and classrooms. In this way, we are sure the user will not have any problem during the

practice, because of a lack of previous information in the *Schoodule* database. Executing this first step of this unit will cause CACTUS to ask the *scenario execution module* to prepare the *scenario*, in order to add default information to the application database. Of course, the user may choose to use her/his own work context instead of preparing the *scenario* provided by the designer. After this, the unit will teach the user how to insert a new teaching assignment to the *Schoodule* database. This teaching activity will be done under the supervision of the Advanced-HATS tutoring component, which will explain the task to the user and will tell her/him whether her/his actions are correctly performed. Finally, once the user would have successfully inserted a new teaching assignment, the tutorial course will give the user some feedback about the task that has been just performed, by means of a feedback message that includes the parameter values of the new teaching assignment added.

When the user adequately accomplishes the pedagogical contents of a unit, CACTUS automatically updates the state of that unit, enabling the access to its succeeding chapters, if appropriate. When a unit is disabled, CACTUS will not permit pupils to execute it. CACTUS shows under each unit title its state: *'Blocked unit*, *'Already performed'*, '*Accessible unit'* or '*Being performed'*. During a unit execution, CACTUS remarks at each moment the instruction being accomplished, so that the user always knows how much s/he has already performed and how much remains to be accomplished.

As previously mentioned, the proposal page is also executable, and its execution means the successive execution, according to the order being shown, of the pedagogical units in the course. By executing the proposal page, CACTUS makes the unit precedence graph transparent to the user, so that s/he will never try to execute a unit whose execution is blocked.

5. Benefits and Future Work

We have described how the model-based user interface design technology can be used effectively to generate dynamic and interactive courses for software applications. The tutorial courses produced with this technology overcome many of the most important problems related to the current systems for tutoring generation.

We have summarized the capabilities of the Advanced-HATS tutoring system that make it a suitable tool for building tutorial courses. This module provides the user with dynamic tutoring, updating the tutoring messages according to what s/he needs to perform at each moment, and offering feedback about what has been previously done, through graphic references to the application interface. Thanks to this module, the system overcomes some of the main drawbacks in current teaching systems. The abstraction level of the explanations is higher, due to the hierarchical nature of the task models the system is based on. The tutor component is more integrated with the application, since it is able to reason about the application interface to provide graphical feedback. Moreover, the Advanced-HATS behavior provides a dynamic style to the explanations. Finally, as the system is based on a run-time task management system, it is able to follow-up user actions and to provide feedback about user activity.

We have also summarized the *scenario execution module*, a subsystem to provide the designers with support for preparing appropriate *scenarios* to practice with. This module also teaches the pupil by showing her/him how to carry out the tasks being performed, and its operation ensures the possibility of teaching the tasks of the tutorial.

We have described the architecture of ATOMS, the user task model run-time manager that provides our course generation architecture with task-oriented support. The use of user task models drastically reduces the maintenance costs of the tutorial courses, since it is almost reduced to maintain the models. We have shown how this task manager faces the generation of the task models by using interactive techniques. It would be very interesting to improve this specification by means of inductive techniques such as grammar induction (Maulsby 1997), or to integrate a more powerful machine learning algorithm with TMT, such as ID3 (Quinlan 1993), to generate more complex models.

We have introduced CACTUS, an implementation of an interactive interface to facilitate the access to the Advanced-HATS tutoring subsystem and the *scenario execution module*. This interface provides automatic support for building tutorials in a fully interactive way, reducing the development costs of the tutoring courses. This interface aims to integrate services oriented both to tutorial course designers and pupils, while isolating the direct access to Advanced-HATS and the *scenario execution module*.

Finally, we would like to port CACTUS to a client/server based platform, at least the part related with the course execution. Our aim is to facilitate the generation of multi-user distributed courses that can be simultaneously executed by several pupils under the supervision of a human tutor who could take decisions at any moment about the future execution of the courses. In this sense, some work has been already done. Thus, (García 1999) introduces a distributed implementation of the ATOMS task management system, which is oriented to manage workflows.

Acknowledgements

Special thanks to Roberto Moriyón for his extensive remarks on the ideas of this paper. This work has been partially supported by the Plan Nacional de Investigación, project numbers TIC96-0723-C02-01/02 and TEL97-0306.

References

Alfonseca, M., García, F., de Lara, J., and Moriyón, R. "Generación Automática de Entornos de Simulación con Interfaces Inteligentes", Revista de Enseñanza y Tecnología, ADIE, nº 10. December 1998.

Contreras, J. and Saiz, F. "A Framework for the Automatic Generation of Software Tutoring". In Proceedings CADUI'96, Computer-Aided Design of User Interfaces, Eurographics, Belgium, June 1996.

Contreras, J. "A Framework for the Automatic Generation of Software Tutoring". Phd. Thesis, Université René-Descartes, Paris, 1998.

Cypher, A. "Watch What I do, Programming by Demonstration". MIT press (ed. A. Cypher), Cambridge, Ma., USA, 1993.

Foley, J.D., Gibbs, C., Kim, W.C., and Kovacevic, S. "A Knowledge-based User Interface Management System". Human Factors in Computing Systems, Proceedings CHI'88, 1988.

García, F., Contreras, J., Rodríguez, P. and Moriyón, R. "Help Generation for Task Based Applications with HATS". In Proceedings EHCI'98, Creta (Greece), September 1998.

García, F., Rodríguez, P., Contreras, J., and Moriyón, R. "Gestión de Tareas de Usuario en ATOMS". IV Jornadas de Tecnología de Objetos, JJOO'98, Bilbao, October 1998.

García, F. and Moriyón, R. "A Framework for Distributed Task Management". Third Argentine Symposium on Object Orientation, ASOO'99. Buenos Aires, Argentina, September 1999.

Johnson, P., Johnson, H. and Wilson, S.: "Rapid Prototyping of User Interfaces driven by Task Models". In Scenario-Based Design: Envisioning Work and Technology in System Development, J. Carroll (ed.), John Wiley & Sons, pp. 209-246. London, 1995.

Maulsby, D "Inductive Task Modelling for User Interface Customization". Proceedings IUI'97: Intelligent User Interfaces. Orlando, Florida, 1997.

Maxwell, J.T. and Kaplan, R. M. "The interface between phrasal and functional constraints", Computational Linguistics, no.4, 1994.

Mitchell, T.M. "Generalization as Search", Artificial Intelligence, 18(2), 203-226, 1982.

Pangoli, S. and Paternó, F. "Automatic Generation of Task-oriented Help". In Proceedings UIST'95, Pittsburgh, ACM Press, 1995.

Quinlan, J.R. "Induction of Decision Trees". Machine Learning, 1(1), 81-106. 1986.

Puerta, A.R. "Supporting User-Centered Design of Adaptive User Interfaces Via Interface Models". First Annual Workshop On Real-Time Intelligent User Interfaces For Decision Support And Information Visualization, San Francisco, January 1998.

Rodríguez, P., García, F., Contreras, J. and Moriyón, R. "Parsing Techniques for User-Task Recognition". 5th International Workshop on Advances in Functional Modeling of Complex Technical Systems, Paris (France), Julio 1997.

Sukaviriya, P. and Foley, J.D. "Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help", in Proceedings UIST'90, ACM Press, 1990.

Szekely, P., Luo, P. and Neches, R. "Beyond Interface Builders: Model-Based Interface Tools". Proceedings of INTERCHI'93, 1993, pp. 383-390.

Vanderdonckt, J. "Knowledge-Based Systems for Automated User-Interface Generation: the TRIDENT Experience". Technical Report RP-95-010, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur. 1995.

Zeiliger, R. and Kosbie, D. "Automating Tasks for Groups of Users: A System-Wide "Epiphyte" Approach, in INTERACT'97 (ed. S. Howard, J. Hammond and G. Lyndgaard), Chapman & Hall Press, IFIP, Sydney, 1997.