Paquette, G. Pachet, F. Giroux, S. and Girard, J. EpiTalk, Generating advisor agents for existing information systems, *Artificial Intelligence in Education*, (7) 3-4,pp. 349-379, 1997.

ÉpiTalk, generating advisor agents for existing information systems

Gilbert Paquette[1], Francois Pachet[2], Sylvain Giroux[1] and Jean Girard[1]

[1]LICEF, Télé-université, 1001 Sherbrooke East St, Montréal H2X 3M4, Canada

[2]LAFORIA, Université Paris 6, E.4, Place Jussieu, 75252, Paris, cedex 05, France

Abstract

Advisor components play an important role in intelligent tutoring systems and learning environments, as well as in electronic performance support systems and help components of commercial applications. A complete advisor should go beyond mere contextual help wired in an information system. It should achieve good balance between the user's and the advisor's initiative. It should be able to offer suggestions at different levels of abstraction, from very domain-specific suggestions to more abstract generic problem solving methods. Finally, it should use various viewpoints on the user's tasks, and support collaborative as well as individual activities. Indeed, this is not trivial.

We present here the research results of the first phase of the ÉpiTalk project. ÉpiTalk is a generic tool designed to facilitate the development of advisor systems. It is based on a design process that formalizes task analysis results in the form of a tasks tree, further structured by progression links between progress levels associated with each task. A model of the application needs to be built and used to define the terms that will be used to trigger the individual actions that the advisor system will provide. These are distributed into advisor agents, each assigned to a specific task in the tasks tree. Each agent has components to analyze the user's activity and select relevant pieces of advise or actions. Then an intervention strategy must be defined and used to select the actual actions that will be proposed to the user.

The ÉpiTalk system has been implemented using a multi-agent architecture. It aims to free the designers from technical preoccupation, encouraging focus on the elicitation of the knowledge needed for the definition of the advisor system. The "epiphyte" property insures independence towards the application, thus favoring reusability and maintenance of advisor components

ÉpiTalk has been used to develop the advisor for a course design workbench (AGD), as well as is two other application domains. These applications demonstrate the feasibility and the generality of the approach. Furthermore various types of advice content and initiative patterns have been implemented. Other issues demanding further research are also discussed.

Introduction

Advisor systems are more and more pervasive in the computer field. We find advisor systems integrated as a central part of an intelligent tutoring system (ITS), as a component in an Electronic Performance Support System (EPSS) or bundled with a text editor or a spreadsheet in the form of a contextual help module or so-called "wizards". With the growing popularity of the Internet, there is a strong need for some form of intelligent assistance to prevent endless browsing through the enormous amount of information available on the global network.

Most of the useful theoretical concepts for advisor systems have been first developed within the context of Intelligent Tutoring System (ITS) research. Essentially, what we call here "Advisor systems" corresponds roughly to the diagnosis and the didactic components of an ITS (Wenger 1987). Based on a diagnosis of the student activity within an application, an advisor system compiles some useful advice or explanation, and delivers them to the student on its own initiative or at the student's request. In intelligent help systems (Winkels 1992) the focus is moved from systematic training or tutoring, to a more opportunistic user-system cooperation for solving problems or performing tasks.

The work presented here aims at providing tools for user-system cooperation in solving problems, whether for training or for task performance. The first section defines our view on advisor systems and states the problem we wish to solve with the ÉpiTalk architecture. The second section discusses the process of designing an implementing an advisor on an existing application. The third section presents the architecture of ÉpiTalk, a task support system for the construction of an advisor. The fourth section gives a detailed account of the use of ÉpiTalk in an extensive application to a course design support system. Finally, the last section discusses the limits of the actual system and outlines the work that is being done on the methodology, on the architecture and on the extension of ÉpiTalk to other task domains.

Advisors on existing applications

In this section, we first define what is meant by an "advisor system". Then we recall previous work on adding advisor components to a learning environment, that have led to the actual architecture. Finally we state our goals and the problems we intend to solve with the ÉpiTalk architecture presented here.

Advisor systems

Let us first define some basic concepts.

—An *advisor* is a system that provides some suggestions, explanations or actions based on the interactions between a user (or a group of users) and a set of tools in an application. As shown in Figure 1, the users and the tools within the application, together with their communication channels, can be seen as a host system on which the advisor will perform its duties.

—An *application* is a coherent set of tools that are not, in general, linearly structured. The user can navigate in that universe without the need to follow a precise route or a sequence and he is not bound by the actions performed with the tools. For instance, a SmallTalk Browser can be seen as such an application. Its windows are linked together in order to present a coherent state of all classes. But it is not linearly organized since the user is free to browse anywhere. Similarly, a law induction system contains tools to define an observation set, to simulate the data gathering process on a physical law, and to process and graph data. The tools are coherent, serving general but precise purposes, but there is no particular sequence for their use.

—A *piece of advice* is any computer action, which does not disrupt the operation of the studied system (the application). Usually, a piece of advice is in the form of a text, displayed in a window at the system's initiative or at the user's request. In some circumstances, the piece of advice will be a more significant system's action, such as running a new tool or presenting a document. However, in our architecture, an advice is neutral with respect to the application, in the sense that the user keeps the responsibility to decide for the course of future action. A counter example would then be an error message followed by the interdiction of a user action. Such integrity constraints, in our view, should be programmed in the application, not in the advisor, because they are essential parts of the tool's definition.

Figure 1 - Advisor on an host system

Four dimensions of intelligent assistance are of interest to us: the user input to diagnosis, the initiative of the user-advisor interaction, the content of the system's assistance, and its capability to assist in collaborative activities.

1. An advisor should go beyond mere feedback and simple contextual help wired in an application. It should have diagnosis capabilities based on the user's prior productions and interaction patterns within the application.

2. An advisor should achieve a good balance between the user's and the advisor's initiative. Learning and task support will benefit from a true cooperation between the user and the system. Too much initiative from the system, and the user has no more room to learn or act. Too little advising and most users will circle around in unproductive ways. An example of such a good balance is ERMA (Brahan, 1992), where an advisor system has been added to SILVERRUN, a conceptual data modeling tool. In its reactive or passive mode the system will help the user only on demand. In its pro-active mode it will monitor its interactions with the user, offering suggestions to improve the final solution on its own initiative. Finally, in its tutoring mode, the system will guide the user through problems for which it already has the solution, aiming at teaching the knowledge domain, that is conceptual data modeling.

3. The content of the advisor's intervention should provide specific domain content as well as some generic methods for the solution of a given class of problems. WHY (Collins 1977) is an early example of a

socratic tutor based on content, favoring the guided discovery of concepts and facts on South American geography. SMITHTOWN (Shute, 1986) is an example of a methodological advisor on the planning of a simulated city. There, an advisor suggests some experiences to help the learner discover new economic relations and some methods to collect data and to analyze it systematically.

4. An architecture for an advisor system should also be applicable to collaborative environments in which more data on the interactions are available while the number of possible interaction patterns with the applications is increased. While most advisors, whether in the ITS or the EPSS fields, have been designed for a one-on-one person- machine interaction, work is active on the very important collaboration issues between learners or users. For example, in a distance learning support system, an advisor system will suggest individual steps to a learner as well as some collaborative interactions with the rest of the group on a computer network (Paquette, 1995).

The LOUTI advisors

In the ITS field, most advisors are closely integrated with the application they support. This is the approach we have first taken within the LOUTI project. In this first stream of research, we have built four advisors for education and training in very different domains such as the induction of scientific laws (COPERNIC), the planning of projects (PLANIF), the selection of a physiotherapy treatment (SONODOSE) and the construction of a taxonomy (LINNÉ) (Paquette 1992a,b).

All these advisors have been built in the same integrated framework. LOUTI is a computer workbench supporting a designer in the process of building a learning environment. In each environment, the learner is provided with a "workspace" in which he/she can build knowledge with tools useful to a generic task, thus building in his mental model the generic knowledge embedded in the tools.

First, a knowledge building shell is assembled by a designer from a predefined tool library to create a customized application. Then, one or more knowledge bases can be added to provide a problem definition together with some information on the problem domain. In general, different knowledge bases can be selected by the teacher or the learner to provide a series of more and more complex problems. Finally, an advisor can be added to this environment using a tracing tool on the learner's interactions.

Each advisor takes the form of an additional tool in the library. Thus it is made available to the learner by the same kind of designer's action and becomes available as an option in the menus of the user environment. These advising tools are generic in the sense that they apply to a large class of domain knowledge sharing the same problem type. As an example, the advisor tools in COPERNIC have been developed with astronomy problems in mind, but they have been used in different domains such as the chemistry of gases, kinematics or electricity. They are useful as long as law induction problems [Langley et al., 87], are concerned but they provide advice using the domain-specific terms defined in the knowledge base.

The LOUTI advisors share the following properties:

—The advisor tools are passive or reactive: advice is displayed only on the learner's demand. A metaphor that could be used here is that of a student raising his hand and receiving specialized advice from the chosen advisor. This is a direct consequence of a constructivist approach to learning (Papert 1980) we have adopted in the LOUTI project. But it is also an important limitation of an advisor's intervention capability.

—The advisor tools decide on the suggestions they can provide using one or many of three sources: the knowledge base constructed by the learner, the expert knowledge base integrated in the environment by the designer, or the events knowledge base (EKB), an abstracted view of the learner's history of previous interactions. In COPERNIC, the advice is based only on the learner's knowledge base. In PLANIF and SONODOSE, a comparison overlay between the learner's knowledge base and the expert knowledge base is used to state the advice. LINNÉ is the only environment to make use of the three types of knowledge bases. In it, two advisory tools use the EKB to evaluate the learner's constructions in the knowledge base and to comment on his/her use of the tools provided by the environment.

—To build the EKB, a "set trace" method had to be inserted in most of the predefined tools in LOUTI's librairies. This method returns to the EKB the modifications done by the learner to the problem's knowledge base on the tools he has used together with the objects to which those tools were applied. A trace tool was then constructed and added to the tool kit in LOUTI, thus providing for its use in any learning environment. This tool uses the "set trace" methods to create an event every time a method of this type is triggered by an object of the learning environment. For instance, if the elements of a chart are sorted according to one attribute, the corresponding event in the EKB will contain the name of the event, the elapsed time since the beginning of the session, the identification of the set that was sorted, the name of the tool (for example sort) and the action performed (for example, sorting on column B variable). This EKB is global, grouping in a single class all the events that took place since the activation of the trace tool. However, it can be manipulated, for analysis purposes, with the same tools that are used in the learning environment (subsets tables, inclusion graphs,...) This kind of analysis can be performed by the learner, the teacher or a computerized advisory module, but this last possibility has not been implemented.

—In all four environments, whichever knowledge base are used, each advisor tool is built according to a preconception of its future role. It uses the KBs to build an evaluation of the learner's work based on his productions (learner's KB), on an overlay on the expert knowledge base (comparison between learner's and expert's KB) or on the events history of the learner (EKB). As many advisor tools as needed for a generic task can be build this way and added to the LOUTI system library of tools and to specific environments.

Goals for the ÉpiTalk architecture

The LOUTI project just outlined has pointed out productivity, reusability, maintenance, and extensibility issues, together with the importance of a more precise model of the advising process. It became the basis used to define the goals of a new project and a new architecture: ÉpiTalk. We will now state the issues addressed by we are the ÉpiTalk architecture.

1— To provide designers with a productivity tool that eases the design of an advisor on an application

Designing a comprehensive advisor component is a complex task that usually involve different specialists: content experts, knowledge engineers, didacticians and computer scientists. Because of the growing importance of advisor systems, a methodology must be developed and integrated in a task support system. We aim here to provide the application builder with some graphic interface for advisor design, minimizing programming as much as we can. This reduction of technical complexity should help the designer to focus on her knowledge elicitation task.

2—To foster the reusability and the maintenance of advisor components on an application.

Applying sound software engineering principles will help maintain and reuse advisor components. We will then adopt the approach of defining an external advisor as a program that can be added to any application composed of task support tools. An advisor that is closely integrated within an application, like in the LOUTI project, increases the difficulty to maintain or reuse it. A change in the advisor often leads to rewriting part of the application. Furthermore, it is not reused easily for similar applications. Finally, it requires a knowledge of the source code of the application that is not always available, or else demand the intervention of a programmer that can analyze and understand the source code when it is available.

3—To ensure the versatility and the extensibility of the architecture.

The architecture of the ÉpiTalk system should provide at first a general framework facilitating the different decisions that an advisor designer has to make; in particular it should provide the possibility for multiple viewpoints on a task and different levels of abstraction in the advice given to the user. It should be extensible to different approaches for student diagnosis and modeling, to a variety of initiative patterns on the active-passive continuum; it should also make way for content specific help as well as methodological coaching for a generic task; and, finally it should provide support to individual learning scenarios as well as for the many collaborative scenarios that can be encountered in a distance learning environment.

4—To provide a framework for research on the advising process

Because the individual advisors will be developed using a methodology embedded in the ÉpiTalk support system, it will enable researchers to go beyond immediate utilitarian concerns. Building a system such as this one is essentially an endeavor in the basic understanding of the advising process. A better knowledge of this activity can in turn have its own larger set of applications, whether ÉpiTalk will be used or not in advising activities.

A design process for advisor systems

We will now begin by describing informally the process by which an advisor can be constructed to support a task within an existing application. The process presented in Figure 2 is viewed from the designer's eyes, as a general method to be embedded in the ÉpiTalk architecture.

Figure 2 - The construction process for advisor systems

Imagine a designer who wants to build an advisor for a given task to be achieved using a common spreadsheet software. She will need to consider the tools provided by the computer environment: table and graph definition, sorting capabilities, database operations... Advising may focus on many aspects. Indeed pieces of advice could be partitioned. Pieces of advice may concern the use of the application by itself; they can also address the task that is intended by using the application. Each point of view gives rise to a distinct advisor. For instance, the advisor that focuses on the application can be reused in many set-ups, while those that are more task specific could not. She will need also to choose a viewpoint on the application: an advisor on a budget task will be quite different from one to support a scientific law induction process; some spreadsheet functions will be useful for the budget task, but not necessarily for the law induction operations. Even within a general task such as this last one, the designer can vary her viewpoint according to the kind of induction task the user will have to achieve. Then, when the viewpoint is chosen, the designer should consider different user scenarios. These are all informal inputs to the process, the output being an advisor system for each chosen viewpoint.

When the designer starts to formalize the advisor, her first steps will be to name and comment the main task (for example "induce a simple physics law from a set of data"), then decompose the task into sub-tasks (for example "obtain data sets", "analyse a table of the data", "graph variables", "state an hypothesis"), and then decompose each of these into sub-tasks until "terminal tasks" are encountered, corresponding to precise actions that the user can make in the application (for example, a sort on a table to compare two variables). The *task tree* built in this first phase is the backbone of the design process.

Once the designer has completed such a task tree, she should know what tools and what kind of interaction with the application are relevant to the task and its possible user scenarios. These are the interactions that have to be reified and diagnosed upon by the advisor. This second phase leads first to a model of the application, or more precisely to a model of the interesting part of the application for the task and for the advisor system on the task. Then, to each sub-task the designer can add a context that describe progress levels within the task. For a given task, there is an on-going activity necessary to perform the task from the beginning to its completion. Progress levels decompose this activity into ordered stages. Each of these progress levels is an abstraction of a diagnosis condition. These conditions refer to the state of the application but mostly to the user's productions. The whole set of progress levels for all the tasks defined in the tasks tree stands as the user model (as overlay models do). These progress levels are any number of designer's defined symbols, the definition using symbols from the model of the application. Finally, the task scenario can be formalized as links between tasks (at a certain progress level), adding a progression relationship to the initial task tree. This is one way to give an advisor some diagnosis capability. At any time, the user can be modeled by its progress level within each task of the task tree.

Then the intervention of the system will rest on an expertise that has to be identified and formalized from the concepts, procedures and problem solving strategies of the task domain or problem type. Some of this expertise can focus on the progress that has been made in a task; that is progress levels associated to each task in the user's model. Another part can evaluate the coherence of the results obtained by the learner by comparing it to some coherence norm. The formalization of this expertise can take the form of a set of individual advices, each with a pattern detection part and an action part (presenting a message, displaying a tool,...) to be performed when a suitable pattern is detected.

Each of these possible pieces of advice can be associated with the proper level in the tasks tree. Lower task in the hierarchy will hold concrete messages such as "you have not yet used the sorting tool on a the following data set: … ", while task higher in the hierarchy will hold more abstract messages such as "you have started validating an hypothesis that is based on very few observations". At the root of the task tree, the designer should assign pieces of advice on the general coherence of all the sub-tasks results. A structure of the advisor system where the pieces of advice are distributed on a tree with a structure similar to the task tree is also good way to cope with different levels of abstraction in pieces of advice. This we will call the *advisor agents tree*.

Once an advisor tree is defined, many pieces of advice can be given in most situations and a choice will have to made to prevent flooding the user under too much unsolicited advice. In other words, an advising strategy must be defined. For example, one strategy could simply be to present the first advice that was detected by the advisor system. A more intelligent strategy would be to give concrete pieces of advice first and, if none are available for a given task, than give any more abstract advice from the advisor agent corresponding to ancestor nodes. Other strategies could differentiate between beginners and more trained users, shifting gradually the emphasis from concrete and strongly directed advice to more general and broad view suggestions. Adding an advising strategy will complete the construction of a prototype of the advisor system. This prototype then will have to be validated with target users to yield the final advisor.

The ÉpiTalk architecture

The ÉpiTalk architecture aims to support the construction process for advisor systems just outlined. It is based on reflection [Maes 89] [Giroux et al. 93] and on the "ecosystem paradigm" (Giroux, 1993) which is a way to define multi-agent systems. In this paradigm, a computer system evolves in accordance to its interactions with its environment. The structure of this ecosystem is defined in a meta-level in which are specified the laws governing the inferior level. This approach is well suited to support pieces of advice both in individual and collaborative activities.

A three level multi-agent architecture

The proposed architecture is composed essentially of three levels. The first level is the reification of the learning environment or application from a certain viewpoint. The second level observes the first one and can provide advice or take some adaptive action to change elements in the learning environment. The third level controls the second and decides when an advice will be given to the user.

At the first level we find the generic learning tools and the tools specific to the field of application together with a model of the task to be advised on, namely the model of the application and the task tree with a progression relationship. The model contains symbols that represent those tools and their functions that allow the learner to solve a problem or to perform a task. For instance, in a chemistry inductive learning environment, this level will contain a representation generic tools such as a simulator and a spreadsheet allowing to conduct experiments and analyze the results.

The second level uses the model of the first level's structure as the base for organizing the information used by the advisor system. Its main component is the advisor agents tree. The spying objects indirectly observe a learner through the actions and parameters that he transmits to the host system. The resulting trace is then stored in the terminal advisor agents memory that they can use to trigger very specific pieces of advice on terminal tasks.

Then, these low level traces can fuel their parent in the tree with proper information, enabling it to constructed a more compiled (abstract) trace memory that it will use to trigger his own pieces of advice and then transmit the compiled trace to his own parents, and this until the root of the advisors tree is reached. This way, it is much easier to separate the level of the user’s activity and the reasoning on it. Many kinds of trace analyzers can be build, or selected from a predefined library, to serve as memory compilation components for advisor agents. For example, some can look for progression levels reached within a task, others can build coherence indicators between tasks.

Figure 3 - ÉpiTalk Architecture

The third level of the architecture is more concerned with the many ways to provide advice. It holds the general advising strategy of the advisor system. It adapts its actions based on a model of the user distributed across agents evolving at the second level. By analogy, we can say that this level will give advice on how to provide advice. It should permit the implementation of principles such as:

- "the learner should not be buried under pieces of advice"
- "at a given stage, the progress of the learner should be monitored more closely"
- "it is best not to repeat the same advice with the same words"

Examples of ways to do this will be given in the last section.

The "epiphyte" properties of the advisor system

The advisor system which is built over a learning environment or a task support system, must be independent from those applications. This independence should be understood in two ways:

- In a conceptual sense: the advisor systems' architecture is in no way similar or isomorphic to the application it is defined on. For instance, the links between tools are not similar to the hierarchy of the advisory agents which are there to "spy" and "reason" on those tools.

- In a software engineering sense: tools should not be designed in accordance to the needs of the advisor system. This constraint is important because it guarantees the reusability of the tools. Any tool designed to fit a particular advisor system looses that important property.

All those properties are nicely summarized in one adjective found in the botanical world: “epiphyte”. This adjective describes the behavior of certain plants that live on top of other plants without disrupting their normal behavior (unlike parasites).

> *epiphyte*: (Gr. "epi", upon + "phyte", plant) Bot. A plant which grows on another plant; usually restricted to those which do not derive nutrition from other plants.
>
> *parasite*: (Gr. "parasitos", lit. one who eats at the table of another) Biol. An animal or plant which lives in or upon another organism (its host) and draws its nutriment directly from it.
>
> (Oxford international dictionary)

Basic principles

The large number of expertise levels involved here naturally lead us to choose a multi-agent architecture (Gasser, 1991). The multi-agent aproach is particularly well suited to the nature of the interactions in open environments, especially if they have to support collaborative work. The advisor system is viewed as a collection of advisor agents, each in charge of a particular task. Some agents are in charge of advising on a very precise task, others take care of a higher level task, parent to a group of tasks. We will now specify the role of these advisory agents, using the four following basic principles:

1. We assume the existence of a tasks tree which describes the tasks of the user in a hierarchical manner. This graph is the main element of the architecture.

2. The advisor agents are also organized in a hierarchical graph. This advisory agent's graph is isomorphic to the tasks graph specified by the designer, in the sense that the decomposition relationship is preserved.

3. The interactions between the user and the tools are collected by software objects called "spies" which are inserted into the system without disrupting its operation. These spies can detect any interaction between the user and a tool and send the appropriate messages to the "terminal" advisory agents. It is the designer who has to specify which particular interactions (messages or events) each "terminal" advisor is in charge of analyzing.

4. Advisory agents pass-on the information from bottom to top. Only the terminal advisors (associated with tools) receive information on the interactions of the user with the tools. Each agent processes this information and transmits it to the superior level, and so on.

These four principles underly the entire architecture. The first two principles describe how the advisory agents are created and organized among themselves. The third principle defines how the interactions of the user with the application are captured. The fourth principle specifies how the information is transmitted from an advisory agent to another.

Describing the application, the tasks and the advisors

The architecture is based on the manipulation of three representations based on graphs. In the current SmallTalk implementation of ÉpiTalk, there are tools to view and edit the representations, and to point the information to spy upon.

The application is represented by the graph of the running tools

This simple graph represents the current state of the application. It shows the tools currently running. Each time a tool starts running, a representation of that tool is added to the graph. Nodes in this graph represent tools which are considered "real tools". Connections between the nodes represent connections between the tools. For now, connections between the nodes are not all depicted because they do not all step in the construction of the advisor system. The window of this graph is only used to show the state of the application.

A point of view on the task within the application is represented by a tasks tree

The second graph is the most important: it is the tasks tree. This graph abstractly specifies the task, and implicitly, the advisory system's architecture. The hierarchical nature of tasks trees provides the right handles to achieve advisors generic with respect to hosts, operating systems, and platforms. Such genericity shelters advisors from rapid software evolution. The basic idea is to partition the knowledge graph into spies, trace analyzer and advisor layers. The spies layer collects message or primitive events. The trace analyzer layer combines primitive events into higher level ones. The advisor layer processes higher level events to produce pieces of advice. For instance, we designed a tasks tree that was reused for two different word processors: Microsoft Word for Windows 3.1 and a homemade word processor implemented in Smalltalk on a MacIntosh. For the latter host application, the trace analyzer was implemented as a multi-agent system according to the message spying principles given below. For the former, it is implemented as a syntactic analyzer for DDE based on a grammar[1]. As advisors are designed, libraries of high-level events and translators are defined and enriched.

[1][Ritter and Koedinger, 95] proposes an approach similar to this solution.

Figure 4 shows the tasks tree editor in ÉpiTalk. It helps the designer to build, modify, save and update the task and the links in the tasks tree. Terminal tasks (or leaves) are in brackets. Starred tasks have multiple sons instead of a single son (on the Figure, the Browse* task can lead to open a certain number of browsers).

Figure 4 - Tasks tree editor

The "task leaf editor" shown on Figure 5 allows the designer to specify which classes will be spied upon by the appropriate advisor and, for each class, which messages will be captured. This editor's window includes three lists: a list of the SmallTalk classes that need to be spied upon (this list can be reduced or expanded as tools in the application become or stop being active); for each class a list of messages in that class that can be spied on; and, finally, a list of the already selected messages for the selected class.

Figure 5 - Terminal tasks edition window

The spying of the interactions is based on the following two hypotheses:

1. After a user action, messages transmitted to objects are spied on.
2. Messages are spied on at the reception (and not at the emission).

For instance, a mouse click will not be collected when an item is selected from a window. What will be collected is the message sent to the object by that window to signify the item has been selected.

Technically, in the actual SmallTalk implementation, this is done in the following way . A spy agent S on a Smalltalk object X is created right after X is created. S will now receive all the messages that were previously sent to X by other objects in the application. The spy S will forward these messages to X so that the application continues to function undisturbed. But S will also send a copy of the messages to the proper terminal advisors so that the advising process can start. (Pachet et al, 1995)

This mechanism is an important improvement from the previous LOUTI architecture where "set-trace" methods had to be manually inserted in the code of the application, creating many limitations on the scope and the possible reusability of advisors built on the application. But it also has some non intuitive consequences (minor to our view) such as the following

• The only collected user interactions are those resulting in an object receiving a message. This may seem trivial but the system will not be able to spy on those clicks or user actions which are not interpreted by the application, for instance, an inactive key.

• Certain messages will not be captured by the system, for instance messages sent by an object to itself. This is coherent with the notion of spy, which intercepts communications between different agents. Internal activity cannot be spied on. The advisor remains outside the application.

The advisor agents tree is isomorphic with the tasks tree

The third graph is the advisor agent's tree. This tree is automatically generated by the system, together with the "spy" agents, as a by-product of the tasks graph and the edition of spies on the terminal tasks.

The basic premise of the system is that the advisor agents tree is isomorphic to the tasks tree. Isomorphism is achieved simply by associating one and only one advisor agent to each node in the tasks tree. If the node is terminal, the advisor agent will also be terminal. In addition, the hierarchical links of the decomposition tree will be carried into the advisor's tree.

The semantics of the advisors' graph is defined by the tasks tree as specified by the designer. It is indeed from the tasks tree that the advisor system will be constructed, when the application starts running. More precisely, the tasks tree has two functions:

• to organize the advisory agents into a hierarchy;
• to specify, for each terminal advisor, which tools should be monitored, and for each tool, which messages should be intercepted.

Thanks to the hierarchical nature of knowledge graphs, diagnosis (plan recognition) and advising are performed in a single walk through at run-time. Different alternatives thus correspond to different tasks agents tree that operates concurrently. There are also mechanisms that detects implicit steps in the course of actions. (Leman et al, 1995)

Components of an advisor agent

We have described how the advisor agents are organized with respect to one another. We will now describe the operation of one advisor in particular. This is governed by the two following principles:

1. Only terminal advisors receive information about the user’s interactions.
2. Information is passed on from bottom to top in the advisors graph.

Specification about message analysis for a terminal advisor is done through the task leaves editor. Each terminal advisor may receive information from a number of tools and the messages to intercept can be specified for each tool. When the information is received, the agent, whether terminal or not, processes the information and passes it on to its hierarchical superior.

Local treatment of the information follows a simple pattern: each agent is created with predefined components, each being specialized in one particular type of processing. Local processing simply consists of sending the information to existing components within the local advisor.

Information transmitted to the hierarchical superior of an advisor agent is not necessarily identical to the one received by the agent at the lower level. It can be more abstract. The higher in the hierarchy an advisor is, the more abstract is the information processed, and the smaller is the volume of the information processed. Ideally, direct interactions with the application should be only manipulated at the terminal advisors level.

In the advisor agents tree, the information on the user's actions is first sent to the leaves (terminal agents); to process them, the terminal agents send the information to their local components (down). The resulting information is sent to the higher level and from each higher level advisor agent to its components (down), and so on.

Predefined components of an agent

The concept of a predefined component provides the designer with a simple and efficient way to define the behavior of an advisor agent. The idea is to offer a range of standard predefined components, organized in classes. Each predefined component class is capable of performing a small specialized task, such as store a piece of information, find regularities of some type, handle the deletion of an object, trigger a set of rules, etc.

All the designer has to do is to plug, into the advisor, the components that she requires. At first, the agents are created without any components. Consequently, they do nothing. The behavior of an agent can be simply redefined by selecting a class among the existing components, or by defining new ones.

At present, only a small number of components have been defined: a rule base, a collector and two kinds of memorization components; they are accessible to the designer via an editing interface.

—The *Rule base* component enables the designer to directly attach pieces of advice to an advisor. This can be done in a variety of ways. As a default value, a link has been made to NeOpus (Pachet, 1992), a rule base editor within SmallTalk, but in some application, we can implement the rule base component in a different way.

—The *Collector* component is used within terminal advisors for spying the creation/opening of new tools.

—The *Memory* components store the result of the analysis they make for their own local advisor of for parent advisors. Actually, there are two available memorization classes: one that registers all that it receives (AgentMemory) and one that does nothing (AgentNoMemory). Other sub classes should be added which would store information on a selective basis, or which would keep the information only for a given time. Another possibility would be a memory which would attempt to detect "regularities" within itself.

All possible cases cannot be covered by the components available at a given moment. That is why, to help the designer in her tasks, it will be necessary to define new classes of predefined components. The application in the next section will show an example of such an extension. Another interesting possibility for the future is to add components well suited to advice on some generic task (Chandrasekaran, 1986).

An application to a course engineering workbench

Three applications have been undertaken using ÉpiTalk to add an advisor system to a learning environment or to an EPSS. One operational advisor has been built AGD, a course design workbench (Paquette et al, 1994). This application will serve our purpose to illustrate the use of ÉpiTalk methods and tools.

AGD tasks tree

AGD is a task support system intended for content experts who are designing a course or learning activities. The system is based on conceptual, procedural and strategic knowledge specific to the instructional design domain. Didactic engineering tasks, such as knowledge distribution into modules or statement of learning objectives, are situations in which a designer requires some support. The advisors need input from tools corresponding to those tasks, and more precisely on specific actions within those tools such as the transfer of a knowledge unit or the statement of a learning objective. These are provides by the spying agents defined on terminal task (insertion points) of the advisor system.

Figure 6 - Tasks and advisor agents on the AGD application

The upper part of Figure 6 shows the upper layers of the AGD task graph. The main task has been decomposed into 168 sub-tasks, including the task “model and distribute knowledge (into courses, modules and activities)”.

The lower part of Figure 6 show this “model and distribute” task further decomposed and associated to the corresponding agents of the advisors tree. Tasks like create or delete a knowledge unit or a link in the model are terminal tasks, corresponding to direct user actions or set of actions in the interface of AGD.

Using context for advising

We can define progress levels to add contextual information to each task. This information enables ÉpiTalk to build a structured memory of the user’s actions pertaining to the task. Using as an example the “construct the model” task, the following table shows an ordered list of such progress levels. The second column gives the informal definition of each level; in the actual implementation of ÉpiTalk, this definition is formalized in SmallTalk code using the lexical terms in the model of the application. The third column gives the general meaning of the associated pieces of advice. Each piece of advice will be fired if the user is at or further than the corresponding progress level, but has not reached the next level.

“Construct the model” task

Progress level	**Condition**	**Associated pieces of advice (if user is further than that level)**
Not started	Default value at start	**Model-01** "Please start using the model editor....."
Model started	"Knowledge unit list in the user defined model is not empty "	**Model -02,03** "Try to complete an initial model containing at least 10 knowledge units" "Specify the learning needs for all client groups"
Initial model sufficient	"The model has at least 10 knowledge units for which the learning needs have been specified for all the client groups"	**Model -04** "The following knowledge units are not linked to the rest of model: ....” **Model-05,06,07,08,09** "The model is not well balanced for this reason:........." **Model -10,11** "There are not enough knowledge units because you have too many (courses, modules or learning activities)......."
Model completed and well-balanced	Model contains enough knowledge units (a number sufficient to “cover” each course, module or learning activity already defined) **&** Model is well balanced, that is relative % of knowledge units that are facts, concepts, procedures and principles is in accordance to the learning needs of the client groups **&** All the knowledge units in the model are linked to at least another knowledge unit.	**Model -12** "This task is completed for now, you should turn to the development of the pedagogical structure and add some course, module or learning activities"
Mainly achieved	Always false	

Table 1 - Progress levels and pieces of advice on the “construct the model” task in AGD

Notice that the second and third progress levels refer to another task “specify learning needs” that is related to “construct the model”. This is an example of a progression relationship between two different tasks. The present task with its level “initial model sufficient”, should be preceded by the task “specify learning needs” at the progress level where it is “Mainly achieved”: learning needs are specified for all the knowledge units.

Structure of an advisor agent

Each advisor agent on a task is informed by the part of the user model used by its advising rules component. This memory component stores the progress level that the user has reached and also the values of his related productions. Table 2 gives an example of the state of an advisor when a rule such as “Model-05” is ready to fire:

PROGRESS LEVEL IN THE TASK:
“Initial model completed” < = USER < “Model completed and well balanced”
USER PRODUCTIONS:
USER has produced learning needs that are at the SENSIBILIZATION level
USER actual knowledge model contains: Facts = 10%; Concepts = 30%; Procedures 35%; Principles = 30%.
EXAMPLE FROM THE ADVISOR’S RULE BASE:
Advice name: “Model-05“
• *Advice*: “Increase the proportion of facts and concepts in your knowledge model by adding new facts or concepts or deleting some procedures or principles.
• *Additional explanation:* Usually, learning needs at the SENSIBILIZATION level entail mainly factual (more than 30%) or conceptual (more than 40%) knowledge units
• *Progress position:* between “initial model completed” and “model complete and well balanced” levels
• *Additional condition:* Maximum target for learning needs < = 2.5 and (Facts% < 30 or Concepts% < 40)

Table 2 - State of some components of an advisor

Concrete and abstract advisors

If we go a step higher in the hierarchy, the advices on the “model and distribute” task will become more abstract in nature. As shown in table 3, the progress levels of this task are defined in terms of progress levels of its sub-tasks, and pieces of advice are focused on the general progress in task achievement.

“Model and distribute” task

Progress level	Condition	Associated piece of advice (if user is further than that level)
Not started	Default value at start	**ModDistr-01** “Start building an initial model and create a first learning event”

Model-distribute ready to start	Model started & Create pedagogical structure started	**ModDistr -02** "Start developing sub-events of the main learning event and distribute knowledge units in these events...."
Model-distribute started	Model started & Distribution started	**ModDistr -03** "The knowledge model is inadequate for some of first-level sub-events ..."
Model-distribute completed on level 1	Model completed and well-balanced & Pedagogical structure completed on level 1 & Distribution completed on level 1	**ModDistr -04** "Due to the number of knowledge units in your model, you should consider developing a second-level layer of sub-events"
Model-distribute adequate	Model completed and well-balanced & Pedagogical structure completed & Distribution complete	**ModDistr -05** "You should proceed with the "state learning objectives" task"
Mainly acheived	Always false	

Table 3 - Progress levels and pieces of advice on the "model and distribute" task in AGD

Discussion of the results

In this final section we discuss our results covering the following aspects: feasibility and generality of the architecture; versatility and extensibility of the advisors; support to the designer and technical development of the system; intervention strategy and coordination of advisor agents. While stressing the achievements and limits of the present stream of research, we will identify more issues that needs to be addressed, some of them being undertaken in our research group.

Feasibility and generality of the architecture

The advisor described in the preceding section has become fairly mature. In fact AGD is undergoing an extensive validation in six organizations that will serve also to gather data on its advisor built using ÉpiTalk methodology and tools. The fact that we have designed an advisor for such a complex task as course design in a few months is certainly a demonstration of the feasibility of the architecture.

We have also used the ÉpiTalk architecture in two other domains of application. The COPERNIC learning environment developed using the LOUTI system (Paquette, 1992a), has been reconstructed in SmallTalk using ÉpiTalk in an effort to develop a more sophisticated law induction advisor. COPERNIC-2 presently comprises tools to plan an experiment and to generate observations, to analyze information using charts and graphs, to look up for constants, to formulate an hypothesis, and finally to validate and to generalize the hypothesis. The advisor has been produced very rapidly using ÉpiTalk, giving us proof of the generality of the architecture with respect to application domains and problem-solving task.

Another application has started recently to test the applicability of ÉpiTalk to collaborative learning environments. HyperGUIDE is a hypermedia software system being presently developed at Télé-université for distance learning (Paquette, 1995). It is the interactive learner's road map which describes the learning activities and give access to the available pedagogical resources: teachers/experts, co-learners, multimedia documents, interactive advice. The tasks tree in an HyperGUIDE expands on four levels (1) Course level (2) Module level (3) Learning activities level (4) Individual consultation or collaborative interaction and productions level within a learning activity. The terminal tasks consist either of consulting input documents, consulting persons (tutor, co-learner, team, group), or producing homework using tools integrated in the HyperGUIDE, or activated by it. The interesting aspect here is the possibility to build specialized tasks graph for each point of view taken on the learning environment. We can examine the nature of the interactions among learners; or among the learners and the tutor, the knowledge consulted and produced by a learner, or his general learning path within the activity network, the modules and the course. Work on this application has just started, but a three-year program should yield a better understanding of the advising process in collaborative tasks.

Versatility of the advisors

AGD's implementation has demonstrated that the architecture is versatile enough to allow multiple forms of advice.

- Some of the agents of the advisor tree can take care of a small group of user action in the application, while others will look at global coherence between the user’s production, and still others will offer suggestions on more generic problem solving issues. Generally speaking, the lower advisors in the tree will provide for domain specific advice, while higher advisors will focus on generic methods issues. In the AGD application, a study of instructional design has provided some generic knowledge integrated in the more general advisor agents. Another way to combine different advisory contents is to build two or more tasks trees corresponding to different viewpoint on the application and then coordinate the resulting advisors. This possibility provided by the architecture has been tested only on a limited scale.

- The balance between the user and the advisor’s initiative is the responsibility of the designer. Any initiative pattern, from user only to system only can be decided upon. In the AGD advisor, one version was built with control left to the advisor when dramatic issues were at stake. The rest of the time, the advisor would wait for the user to ask for an advice.

- Even though the architecture of ÉpiTalk has been designed to support collaborative environments, this possibility has only been tested in small experiments simulating cooperative activities in a Scrabble game. The new application on the HyperGUIDE distance learning environment will provide the real scale test on this issue. Some very practical spying problems will have to be resolved when many applications such as E-mail, assisted teleconference and groupware are used simultaneously for collaborative work in different locations. Important theoretical issues, especially those pertaining to the role of human coaching ressources vs computerized agents in a collaborative activity will also have to be resolved.

Support to the designer and technical issues

Some designer interfaces such as those presented above help minimize the use of computer programming. However, a good knowledge of SmallTalk programming is still necessary to use them efficiently in the actual state of ÉpiTalk. For now, our applications such AGD and COPERNIC-2 are SmallTalk programs thus simplifying technical problem. A SmallTalk program provides a complete model of the application. It is then easy to exploit the code of the application through a “task leaves editor” such as the one presented earlier, so that the designer can specify what objects and messages will be spied upon.

But this is an important limitation. The designer should be able to build an advisor on a “black box” application, working essentially with symbols she understands, referring to the concepts in the application and the task, and not to their physical implementation.

To go further in that direction, we have recently developed a C++ program that uses the Windows DDE system to spy upon any well-defined Windows application from the outside. Then, adding a parser based on a grammar with a lexicon of Windows messages, we can reconstruct a SmallTalk model of the application using designer’s defined symbols. For example, a generic model for the basic functions of a text editor can be stored and adapted to a particular text editor like WORD or WORDPERFECT. Then this model of the application can be used to define constructs like progress levels within tasks and conditions in individual pieces of advice definition, without references to the physical implementation of the application.

To totally eliminate programming seems difficult. However it would be possible to further reduce its use in the most common cases, by integrating more plug-in components that can be added to advisor agents. In the coming months, we intend to reprogram the ÉpiTalk advisor editor along those lines to give a better support to the designer, freeing her from most programming tasks. We will also develop graphic interfaces more coherent with the advisor design process that was outlined in a previous section.

The advising process and the coordination of the advisors

The coordination of the advisors has to be tackled more systematically than we have until now. At any time, many agents from the advisor tree can fire an advice at the user if no meta-advisor component is present, and the situation can be even worse if there are more than one viewpoint on the application. In the AGD advisor, three methods have been used at this meta-level to coordinate the advisor agents, but they have to be extensively.

- The first approach consists of control tools on the selection of available pieces of advice. These can be used by the designer (and the user) to prevent the repetition of an advice to the same user more than a chosen number of times. We have also made possible to pile up the possible pieces of advice, making available only the one with the highest priority with regards to the actual user task, but giving the user the choice to see more pieces of advice if he wants to.

- The second approach is to use the neighborhood properties of the agents. The default strategy exploits the hierarchical structure of the tasks tree. If the advisor on the actual low-level user task has nothing to say, its parent advisor will be called upon to provide its (more general) advice. But this wired-in scheme can be tailored by the designer; she can link other advisors than the parent, for example those linked by a progression or some other relationship. The next step would be to link alternative advisors using some form of genetic graph approach (Goldstein, 79)

- The third approach is to select pieces of advice according to so-called "pedagogical phases" defined in the user model. The pedagogical phases (motivation, acquisition, performance, feedback...) are ordered. As do progress levels, they define a new dimension along which pieces of advice are classified. There is a partition of pieces of advice along those phases. Some of the EuroHelp coaching strategies could probably prove very useful here (Winkels, 92) but it will have to be adapted to our multi-agent approach.

The ÉpiTalk project is still in its early phases. Compared to the comprehensive research results achieved in a project like EuroHelp, ours are limited by the fact that the task description is not linked to comprehensive support knowledge. As a consequence, our advisors do not possess real tutoring capabilities. Also, full user-system dialogs remain limited. We plan to address these issues in later stages.

Right now, the originality of ÉpiTalk lies in the use of the distributed AI paradigm, the idea of reusable advisor agents components, the distribution of advisor agents on different level of abstraction isomorphic to the task tree, and the coordination of partial viewpoints on an application.

In the coming months, we will add new tools to ÉpiTalk to facilitate the design of advising strategies for advice selection and advice display, at the third level of the architecture of ÉpiTalk. These strategies should be expressed in a way that can be easily tested and refined by a designer without the use of programming skills. In parallel, we plan to generalized the tree representation for tasks to a full knowledge model where tasks, input and output concepts and regulating principles will be linked together to provide more support to the user.

For the next step, we can use the actual ÉpiTalk system as a kind of “boot strap”. Having some users experimenting on an ÉpiTalk application, we can use the actual system to collect data for the definition of advising strategies. The fact that this can be done in different domains, but with the same infrastructure and the same plan in mind, should ensure the generality of the system and improve the genericity of the mechanisms.

While there are still a lot of issues to tackle, we believe that the ÉpiTalk architecture has a great potential for intelligent advising and help systems. It can act as a higher level authoring tool reducing considerably the complexity of the advisor building process, so the designer can concentrate on the more important expertise elicitation tasks. Hopefully this will lead to efficient situated learning environments and task support systems and, at the same time, will help increase our knowledge of the advising processes.

References

Anderson, J.R., Boyle, C.F., Farrell, R.G. & Reiser, B.J. (1984). Cognitive principles in the design of computer tutors. Proceedings of the Sixth Cognitive Science Society Conference, Boulder Colorado, 1984.

Brahan, J.W., Farley, B., Orchard, R.A., Parent, A. & Phan, C.S. (1992). A Designer's Consultant. Technical Paper, Institute for Information Technology, National Research Council, Ottawa, 1992.

Briot, J.P. (1989). Actalk: A test Bed for Classifying and Designing Actor Languages in the Smalltalk-80 environment. ECOOP '89, pp. 109-130.

Browns, J.S., Burton, R.R. & Bell, A.G. (1975). SOPHIE: A step towards a reactive learning environment. Int. Jrnl. Man-machine studies, vol. 7, pp. 675-696.

Chandrasekaran (1986) Generic tasks in knowledge-based reasoning: High level building blocks for expert system design. IEE Expert, 1.

Gasser, L. (1991). Social Conceptions of Knowledge and Action: DAI Foundations and Open Systems Semantics. Artificial Intelligence, 47, pp.107-138, 1991.

Giroux, S. (1993). Systèmes et agents : une nécessaire unité. Thèse de doctorat, U. de Montréal. Aout 1993.

Giroux, S., Senteni, A., Lapalme, G.: *Adaptation in Open Systems*, First International Conference on Intelligent and Cooperative Information Systems (ICICIS), IEEE Computer Society Press, May 11-14, 1993, Rotterdam, Hollande, pp. 114-123.

Goldstein, I.P. (1979). The genetic graph: a representation for the evolution of procedural knowledge. International Journal of Man-Machine studies, 11:51-77, 1979

Langley, P., H. A. Simon, G. L. Bradshaw et J. M. Zytkow, *Scientific Discovery: Computational Explorations of the Creative Process* , MIT Press, Cambridge, MA, 1987.

Leman, S., Giroux, S., Marcenac, P., *A Multi-Agent Approach to ModellingStudent REAISoning Process*, AI-ED 95, Aug. 16-19, 95, Washington DC, USA, pp. 258-265.

Maes, P.: *Concepts and Experiments in Computational Reflection*, OOPSLA '87 Proceedings, Orlando, Florida, October 4-8, 1987, pp. 147-155.

Pachet, F., Wolinski, F., Giroux, S.: *Spying as an object-oriented programming paradigm*, TOOLS Europe '95, France, March 6-9, 1995, Prentice-Hall, pp. 109-118

Pachet, F. (1992). Représentation de connaissances par objets et règles : le système NéOpus. Thèse de lUniversité Paris 6. Paris, Septembre 1992.

Paquette, G. (1992a). Metaknowledge in the LOUTI development system. Proceedings of CSCSI-92, Canadian Society for Computational study of Intelligence, Vancouver, May 1992.

Paquette, G. (1992b). An Architecture for Knowledge-based Learning Environments. Expertsys-92, Paris.

Paquette G., Crevier F., Aubin C. and Frasson C.. (1994) *Design of a Knowledge-Based Engineering Workbench* , Proceedings of CALISCE-94, Paris, September 1994.

Paquette, G. (1995) *Modeling the Virtual Campus*. in “Innovating Adult Learning with Innovative Technologies (B. Collis and G. Davies Eds) Elsevier Science B.V., Amsterdam.

Papert S. (1980) Mindstorms: Children, Computers, and Powerful Ideas. NewYork: Basic Books.

Ritter, S., Koedinger, K. R.: *Towards lightweight tutoring agents*, AI-ED 95, August 16-19, 1995, Washington DC, USA, pp. 91-98

Shute, V.J. &Bonar, J.G. (1986). An intelligent tutoring system for scientific inquiry skills. Proceedings of the Eighth Cognitive Science Society Conference, Amherst, Mass. 1986.

Stevens, A.L. & Collins, A. (1977). The goal structure of a Socratic Tutor. Proceedings of the national ACM Conference, Seattle, Washington, 1977.

Stevens, A.L., Roberts, B. & Stead, L. (1983). The use of a sophisticated interface in computer-assisted instruction. IEEE Computer Graphics And Applications, vol. 3, pp. 25-31, March/April, 1983.

Vivet, M. (1991). Knowledge Based Systems for Education: Taking in account the Learner's Context. Proceedings of PEG-91, Geneva, May 1991.

Wenger, E. (1987). Artificial Intelligence and Tutoring Systems, Computational and Cognitive Approaches to the Communication of Knowledge. Morgan Kaufmann, Los Altos, USA, 1987, 486 p.

Winkels, R. (1992). Explorations in Intelligent Tutoring and Help. IOS Press, Amsterdam 1992, 227 p.

Figures

Figure 1 - An advisor on a host system

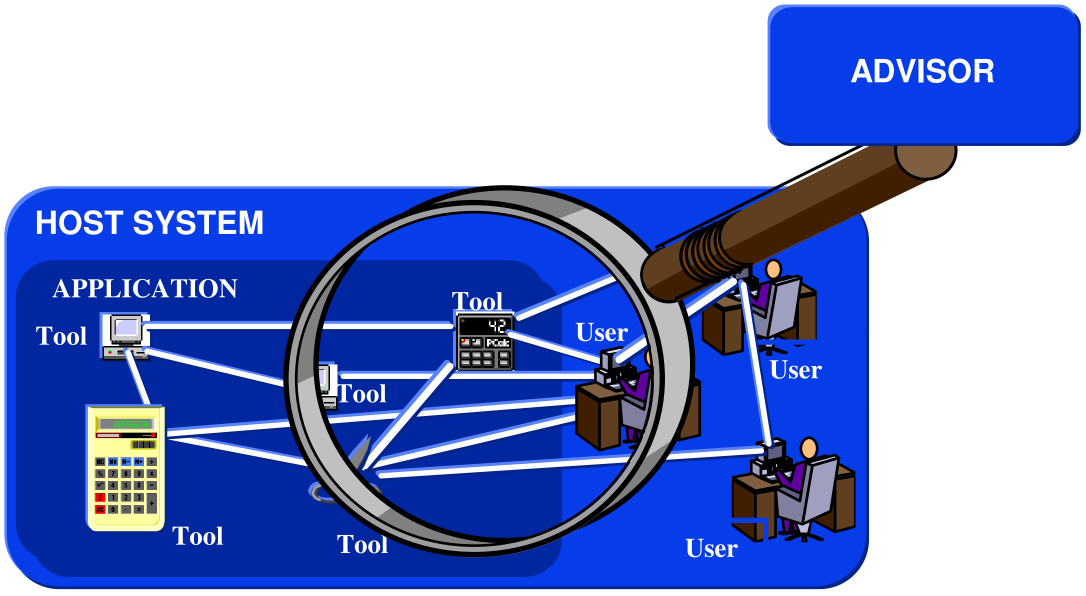


Figure 2 - Advisor construction process

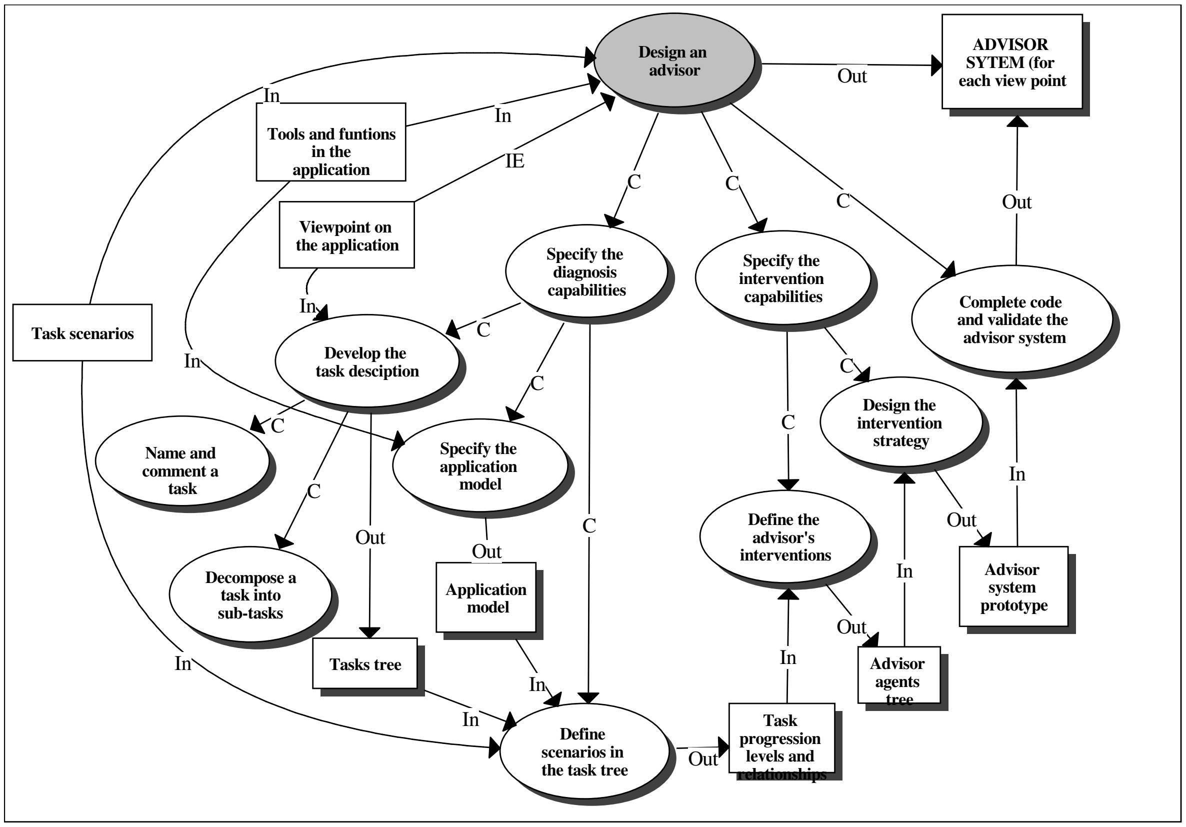

Design an advisor
ADVISOR SYTEM (for each view point
Out
In
Tools and funtions in the application
In
IE
Viewpoint on the application
C
C
C
Out
Specify the diagnosis capabilities
Specify the intervention capabilities
Complete code and validate the advisor system
Task scenarios
In
Develop the task desciption
C
In
C
C
Design the intervention strategy
Name and comment a task
C
Specify the application model
C
C
In
Define the advisor's interventions
Out
Out
Out
Decompose a task into sub-tasks
Application model
In
Advisor system prototype
Out
Tasks tree
In
In
Advisor agents tree
In
In
Define scenarios in the task tree
Out
Task progression levels and relationships

Figure 3 - ÉpiTalk Architecture

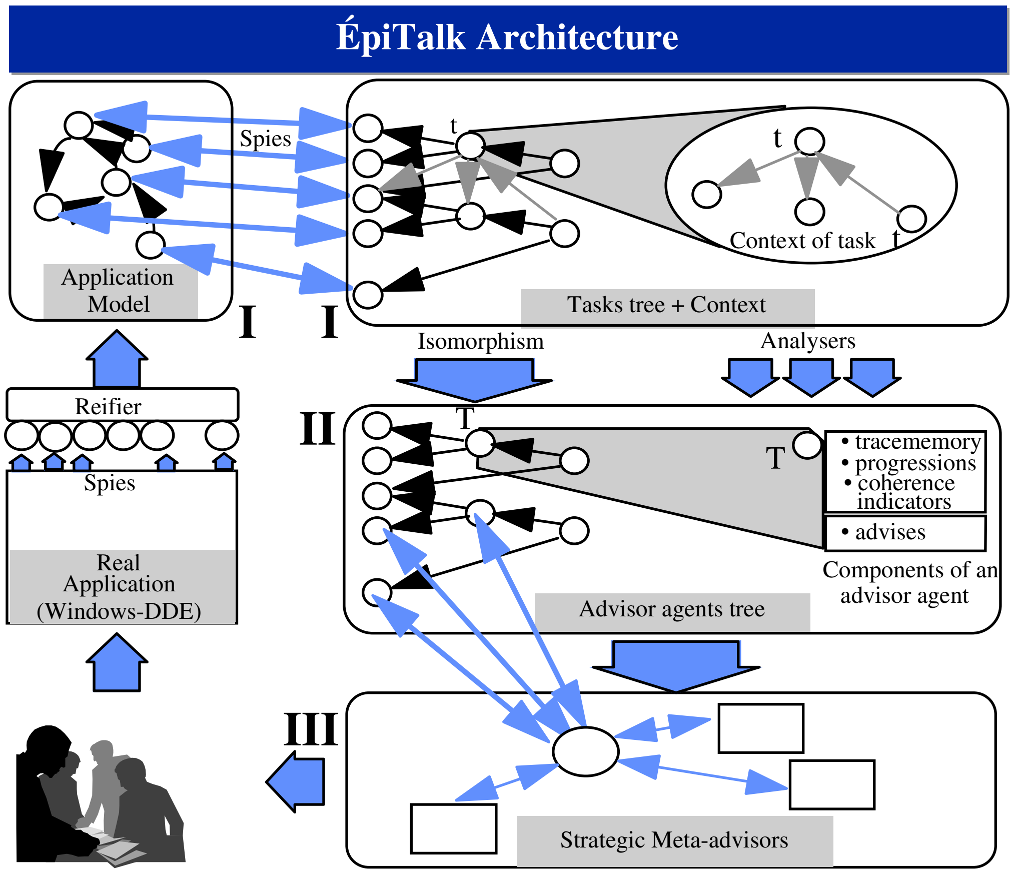


Figure 4 - Tasks tree editor

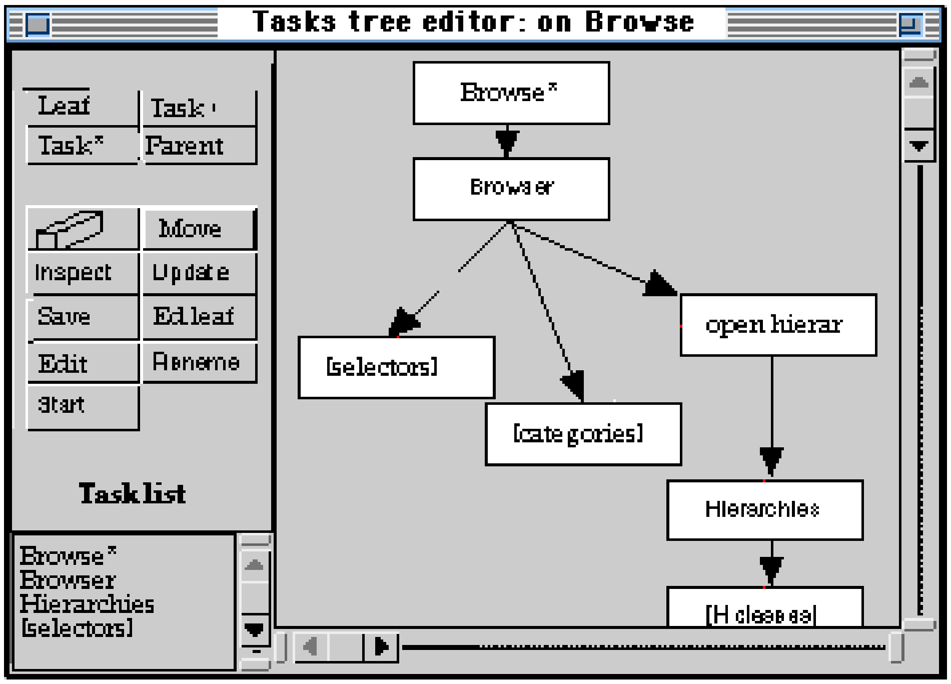
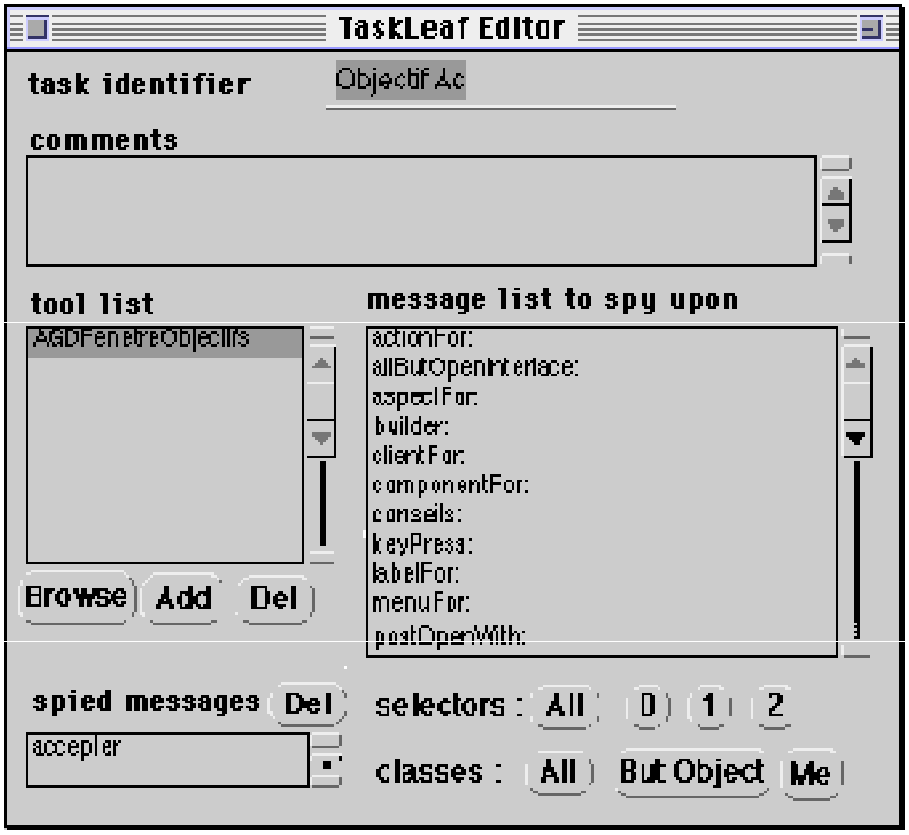


Figure 5 - Terminal tasks edition window



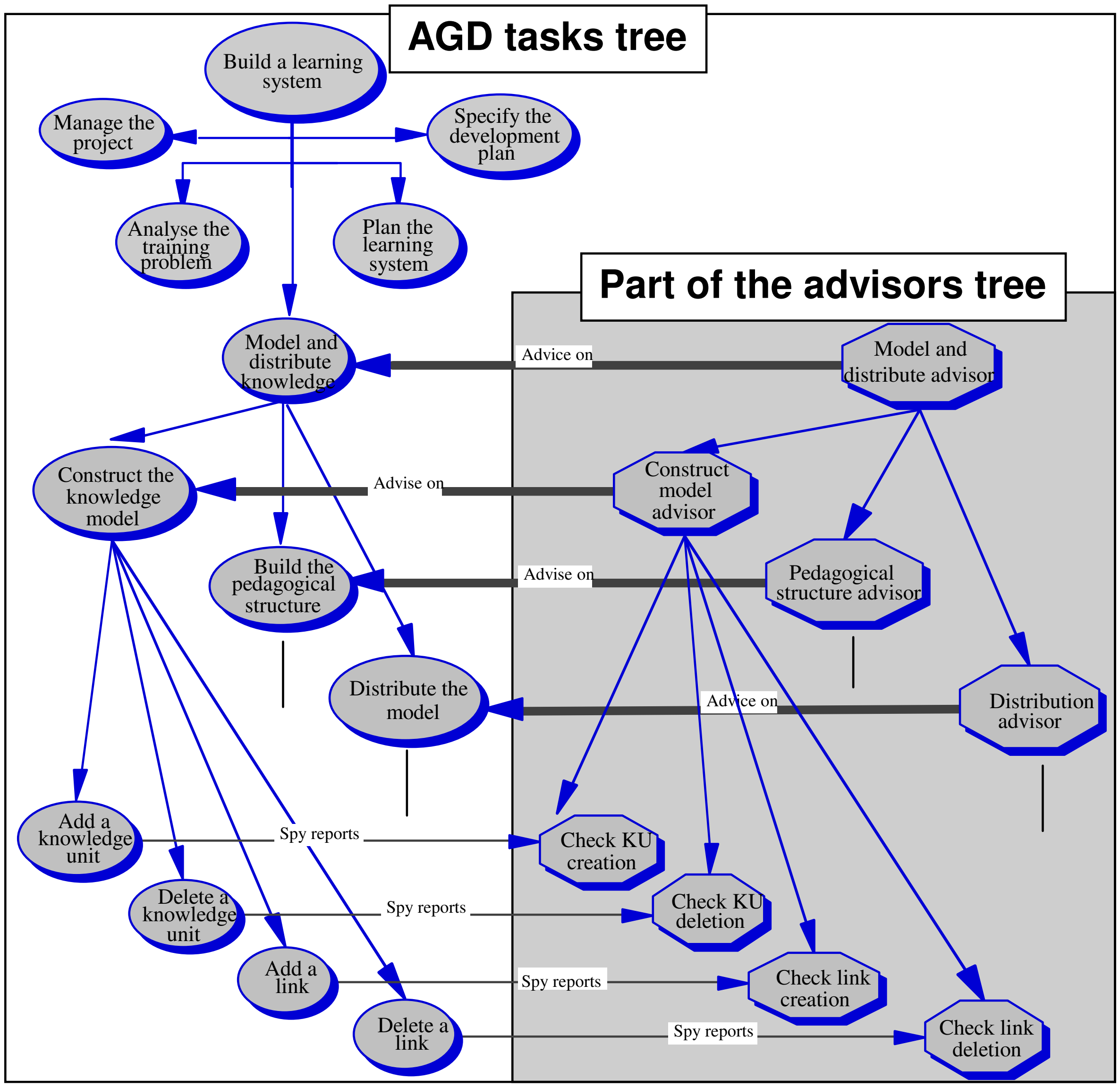
task identifier: Objectif Ac

tool list: AGDFenetreObjectifs

message list to spy upon:
actionFor:
allButOpenInterface:
aspectFor:
builder:
clientFor:
componentFor:
conseils:
keyPress:
labelFor:
menuFor:
postOpenWith:

spied messages: accepter

Figure 6 - Tasks and advisor agents on the AGD application


AGD tasks tree
Build a learning system
Manage the project
Specify the development plan
Analyse the training problem
Plan the learning system
Model and distribute knowledge
Construct the knowledge model
Build the pedagogical structure
Distribute the model
Add a knowledge unit
Delete a knowledge unit
Add a link
Delete a link
Part of the advisors tree
Model and distribute advisor
Construct model advisor
Pedagogical structure advisor
Distribution advisor
Check KU creation
Check KU deletion
Check link creation
Check link deletion
Advice on
Advise on
Advise on
Advice on
Spy reports
Spy reports
Spy reports
Spy reports