Report from the project “Les systèmes explorateurs intelligents”

NFP23 Program; Project No. 4023-2701

Intelligent Learning Environments

P. Dillenbourg, M. Hilario, P. Mendelsohn, D. Schneider, B. Borcic

Abstract

The research work reported here concerns the application of Artificial Intelligence techniques to learning environments. A learning environment is a category of educational software where the learner’s task is not to answer a predefined serie of questions but to explore a complex world with the assistance of computational agents. We implemented MEMOLAB as an instance of such a learning environment. In MEMOLAB, the learner creates a psychology experiment and then asks the system to simulate the results. Our work has concentrated on two directions:

Direction 1: Providing optimal learning conditions. Two main qualities of an environment are its richness and its structure. By ‘richness’, we mean the integration of various tools, for instance the connection between a hypertext, procedural tools (e.g. the simulation) and rule-based agents. The ‘structure’ of a learning environment results from its segmentation into a sequence of microworlds. This sequence should ideally reflect a sequence of learning stages (in MEMOLAB, the neo-Piagetian theory of R. Case).

Direction 2: Providing the assistance of agents. In MEMOLAB, the learner creates experiments in interaction with an expert, i.e. an agent who is able to solve the same problems through the same interface. The degree of assistance provided by the expert is tuned by another agent, the tutor, which monitors the interaction. MEMOLAB includes several tutors corresponding to various teaching styles. These tutors are selected by their superior, called ‘the coach’.

The distribution of roles between the agents has been conceived in such a way that some agents (the tutors and the coach) are not directly concerned by the specific teaching domain and hence can be reused to build other learning environments. The set of domain-independent components constitute ETOILE, an Experimental TOolbox for Interactive Learning Environments. Its originality is that authors do not build a software application by writing questions and feedback, but by designing domain-specific agents that will interact with the agents provided by the toolbox. ETOILE is intended to be used by developers who are familiar with Common Lisp.

TECFA (Technologies de Formation et Apprentissage)
Faculté de Psychologie et des Sciences de l’Education
University of Geneva (Switzerland)
Address: 9, Route de Drize, CH-1227 Carouge (Switzerland)
Contact: P. Dillenbourg
Phone: (+41) 22.705.96.93
E-mail: pdillen@divsun.unige.ch

Table of Contents

1. Goals and Products

The term ‘intelligent learning environment’ (ILE) refers to a category of educational software in which the learner is ‘put’ into a problem solving situation. A learning environment is quite different from traditional courseware based on a sequence of questions, answers and feedback. The best known example of a learning environment is a flight simulator: the learner does not answer questions about how to pilot an aircraft, he learns how to behave like a “real” pilot in a rich flying context. Experience with learning environments (like LOGO) showed that those systems gain efficiency if the learner is not left on his own but receives some assistance. This assistance may be provided by a human tutor or by some system components. In our flight simulator example, the future pilot would gain from discussing his actions with an experienced pilot. The implementation of these agents is based on artificial intelligence techniques.

In summary, we use the word ‘intelligent learning environment’ for learning environments which include (1) a problem solving situation and (2) one or more agents that assist the learner in his task and monitor his learning.

This research project had two main purposes:

- to develop a particular learning environment that illustrates the application of artificial intelligence techniques to intelligent learning environments;
- to develop tools for building similar learning environments, intended to be used by other research teams.

Two software packages have been implemented to reach these goals:

- MEMOLAB is a particular learning environment for the acquisition of basic methodological skills in experimental psychology.
- ETOILE (Experimental Toolbox for Interactive Learning Environments) is a toolbox that enables advanced programmers to build an ILE in another domain, but based on the same principles and architecture as MEMOLAB.

These two systems are described in this report.

2. Research Methodology

Our methodological choices are summarized below:

- ***From the particular to the general.***
  Since we aimed to produce a toolbox and an instance of what can be built with that toolbox, it would have been sensible to build the tool first and then to create an ILE with that tool. We actually proceeded inversely: we started with the design and the implementation of an ILE called MEMOLAB, and we progressively abstracted a toolbox called ETOILE. Many research teams started with ***the*** over-general question: "What functionalities are shared by any educational software?". This approach generates constraints at a high level that unavoidably lead to a set of neutral interface tools and to procedures for specifying the sequence of learning activities. While the authoring process is primarily a pedagogical activity, most authoring tools only take care of the computational issues. By proceeding inversely, we succeeded in designing a generic tool which encapsulates pedagogical knowledge.

- ***A toolbox is not a 'generator'***
  Most authoring tools available on the market focus on designing canonical interactions (question / answer analysis / feedback) and on specifying a more or less complex 'scenario'. These tools use powerful interfaces that enable teachers without programming skills to build courseware. We deliberately avoided the issue of building an authoring interface, because we don't have the resources to produce anything close to the authoring tools being developed by large commercial companies. Instead, our concern was to support a richer interaction than most authoring tools, and especially to create more 'open' problem solving situations. ETOILE provides designers with the bricks of an ILE, but it is the author's task to assemble them into a coherent system. It supports the design process, it does not automate it.

- ***Design is a creative process***
  Current pedagogical and psychological theories cannot be directly translated into system specifications. A particular authoring question raised by a designer must always relate to some psycho-pedagogically valid conceptual framework. This may be supported by providing the designer with metaphors that help him to 'view' his system from some theoretical perspective. During the design of MEMOLAB, we have paid a lot of attention to those ***design metaphors*** that are essential for the conceptual architecture of an ILE.

- ***Propose domain-independent pedagogical knowledge***
  Most authoring tools are pedagogically neutral. For instance, they provide tools to build questions, but they do not include the knowledge necessary to automate the selection of a relevant question. ETOILE includes some pedagogical expertise that can be used for various domain contents. However, pedagogical knowledge is not neutral, i.e. it often reflects particular educational approaches or values. Hence, ETOILE includes multiple pedagogical knowledge bases and it covers a wide and extensible range of teaching styles.

3. Overview of MEMOLAB

Memolab has been built to illustrate the qualities of a good learning environment. The pedagogical domain of MEMOLAB is the acquisition of the methodology in experimental psychology, i.e. experimental design.

3.1. Quality #1: The learner must be active.

MEMOLAB targets the acquisition of the methodology of experimental design in psychology. Nowadays, this methodology is taught in most universities by presenting methodological principles and by discussing experiments from the literature. Students have no real design experience until they have to create an experience with real subjects for their own thesis. At that stage, any mistake in the experimental plan becomes very costly. This strategy for training psychologists could be compared to asking future pilots to fly real aircraft immediately after theory courses, skipping the flight simulator stage. MEMOLAB is an artificial lab where future psychologists can experience (some aspects of) experimental design, without facing the risk of spoiling their subjects. In Memolab, the experimental subject domain is the study of human memory. Basically, the learner's activities form a cycle: design an experiment, run it, and analyze the output data. Here is an example of an experiment that the learner can design: two groups of subjects will study a list of 20 words, and then later on try to remember these words. Between the encoding and the recall, the first group will wait during 10 seconds, while the second group will wait for 40 seconds. The comparison of their recall performance indicates the effect of the delay length. This experiment is illustrated by Figure 1. It illustrates how the learner designs an experiment within the first lab (we will see later that MEMOLAB includes several labs). An experiment is constructed by positioning a set of events on a workbench. The vertical axis represents the time dimension. The activities of a particular group of subjects are vertically aligned: In our example, they first encode some material, then there is some delay and then some attempt to recall what they studied before. When an experiment has been fully designed, the learner may ask to simulate the results. The working details of the simulation are described in Appendix I: "The Simulation" (page 21).

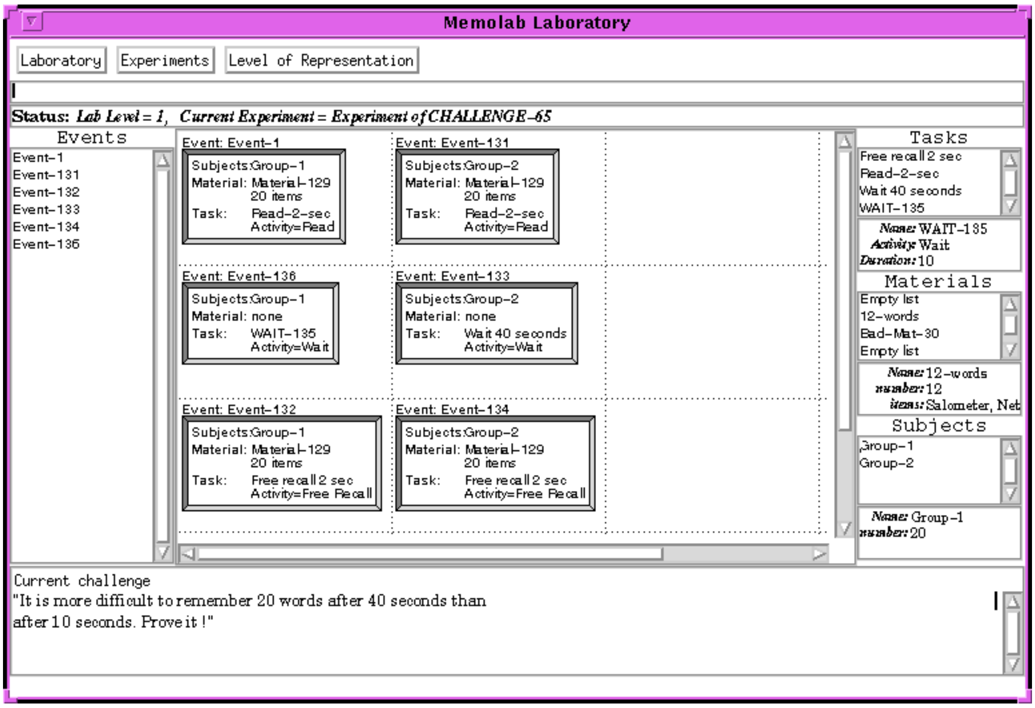

Figure 1: The learner builds an experiment (at level 1) by assembling 'events' on the workbench. An event defines the task (read, listen, recall, wait,...) of a group of subjects using some material (e.g. a list of 12 words). Panes on both sides of the workbench provide libraries where the learner builds and select the components that he assembles on the workbench.

3.2. Quality #2: The environment must be rich and complex

Any psychological or pedagogical theory investigates learning from a specific viewpoint. For instance, some of them pay attention to the effect of practice, but neglect the importance of understanding. Some concentrate on discovery, and forget about imitation. However, when one observes an actual learning process, it becomes clear that learning does not result from a single activity but from the integration of multiple activities, based on multiple sources of information. A learning environment must takes this complexity into account by offering a wide range of learning resources that will accommodate the variety of individual learning styles and learning needs. Therefore MEMOLAB includes several tools:

- ***The make-material tool***
  A difficult aspect in designing an experiment is to select carefully the items that the subjects will have to remember. A list of words has several features that the novice experimenter is not aware of, but which affect memory performance: the length of words (number of syllables), the frequency of words in our everyday language, the phonological or semantic homogeneity of a list of words. Figure 2 shows the "Analysis" tool of the make-material window.
- ***The hypertext 'Methodology'***
  The very idea of a learning environment is that the acquisition of procedural knowledge requires active problem solving (by contrast to questions-based courseware that focuses on declarative knowledge). However, the weakness of microworld designers has been to underestimate the role of domain specific knowledge. MEMOLAB corrects this bias by giving access to declarative knowledge (theories, laws, facts,...) about the methodology of experimentation and theories on human memory. The hypertext structure enables the learner to find information at the level of granularity he is interested in. Our hypertext facility includes the features now common to most similar applications (trace of read nodes, web view, book view, footnotes,...). ETOILE's hypertext module is described in Appendix II: "The Integrated Hypertext" (page 26).
- ***The hypertext 'Encyclopaedia of Memory'***
  Designing an experiment on human memory requires both methodological skills and some knowledge specific to the domain of memory. For instance, learners must know the concept of "indepen-

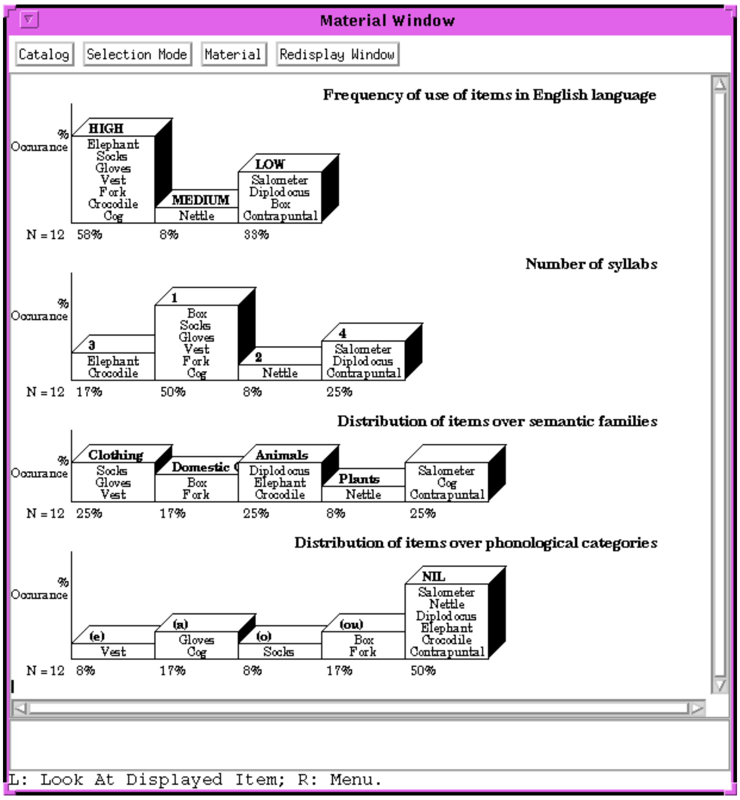

Figure 2:The Analysis of chosen items in the make-material tool

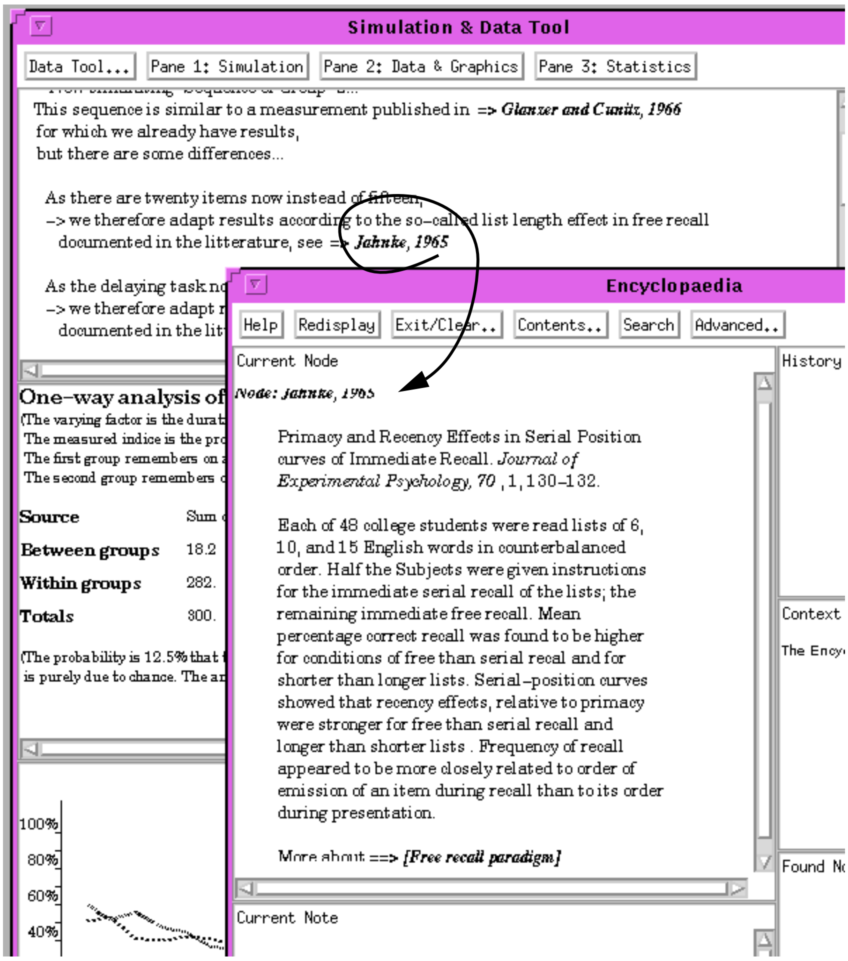

Figure 3:Integration of the hypertext and the simulation: the learner clicks on an experiment referred in the simulation trace and gets the hypertext node where this experiment is described.

dent variable", but they must also know that 'word frequency' is an interesting independent variable linked to 'word length'. MEMOLAB includes an hypertext that provides theoretical and experimental data on human memory.

- ***The simulation & data analysis tool.***
  Research on the use of simulation in educational research (Roschelle,1988; Fredriksen & White, 1988) has shown that the simulation method itself is important. The role of a simulation is not simply to produce realistic results. The simulation process must be carried out in such a way that the learner can understand the results. The simulation built in Memolab produces a commented trace of the simulation process that enable the learner to understand how these results have been computed. Furthermore, a set of statistical results is computed and the learner can inspect the generated data. The simulation tool is described in Appendix I: "The Simulation" (page 21).

The richness of a learning environment is not only a function of the number of tools but of the extent to which those tools are integrated. We made this integration possible by using a homogeneous object-oriented internal representation scheme. Let us illustrate this with the simulation. The results of an experiment are computed by analogy with those of similar experiments (see Appendix I: "The Simulation" (page 21)). MEMOLAB explains how the results have been computed referring to these experiments ('Simulation Trace'). As illustrated by Figure 3, the learner may double-click on one of these references and jump directly into the hypertext node where the referenced experiment is described. The hypertext is also connected with the rule-based agents (see section 4.4. on page 11 and section 5. on page 15). Most current research on hypertext systems has so far been concerned with the features of the hypertext itself. The originality of our approach is to have considered the place of the hypertext within a complex system, to situate the information search within a context that makes this information useful. The architecture of our hypertext is described in

Appendix II: “The Integrated Hypertext” (page 26).

3.3. Quality #3: The environment must be structured

If the richness of a learning environment is a quality, its complexity may reduce learning. The designer must structure his learning environment in a way that the learner may benefit from this richness. The ‘structured environment’ idea illustrates a fundamental difference of perspective between learning environments and traditional courseware: ***the meaning of ‘teaching’ shifts from asking questions and providing feedback (although this is not excluded) towards structuring the environment in a way that provides optimal learning condition***s.

The structure we have adopted is a ***sequence of increasingly complex microworlds*** (or problem solving situations, or laboratories for Memolab). This sequence leads to a “final” microworld that should be as close as possible to the real context in which the learner will have to apply the acquired skills. For instance, the final lab in Memolab uses the language that is commonly used by the community of experimenters in their exchanges (papers, conferences,.). However such a microworld is probably too complex to be mastered directly by a naive learner (as in the real world). It should be preceded by simpler microworlds. In the first one, the learner will have to solve simple problems, considering only some basic parameters. In the next one, he will manipulate more powerful operators and to use additional resources available in order to cope with more complex problems.

The architecture of MEMOLAB includes three microworlds, referred hereafter as ‘levels’. Each level concerns a different kind of experiment and uses a partially different interface:

- At level 1, an experiment compares n groups of subjects along one dimension. The interface emphasizes the concrete, elementary and chronological aspects of an experiment.
- At level 2, the learner describes an experiment as a set of sequences. A sequence is a generic treatment that can be replicated several times with equivalent materials. The learner designs experiments that test the interaction between two parameters.
- At level 3, the interface reifies the logical structure of the experiment as a plan.

By using a sequence of microworlds, one must address the issue of the learner’s transition between microworlds. To smooth this transition, we designed the interface in such a way that these microworlds partially overlap. Each microworld uses a different set of commands for creating and for modifying the components of an experiment (events, sequences or plans). When an experiment has been completed at level N, the system translates this experiment into the representation that it would have at level N+1. The future representation scheme is therefore systematically associated with the actual scheme mastered by the learner. After some time, the learner is invited to express himself directly with the new language (N+1). This principle has been called the ‘***language shift***’ (Dillenbourg, 1992) and is described more completely in Appendix IV: “The Design Metaphors” (page 33).

We used the image of the ***pyramid*** in order to emphasize the hierarchical relationship between operators that are used in successive microworlds. This notion of hierarchical integration is based on the neo-piagetian theory of Robbie Case (1985). In brief, this theory states that the qualitative improvement of children’s cognitive structures is impaired by quantitative limitations in controlling elementary structures. To by-pass this limitation, children restructure their knowledge into higher order schemes that recombine patterns of previously existing sub-schemes. The relationship between Case’s theory and MEMOLAB is described in Appendix IV: “The Design Metaphors” (page 33).

The pyramid metaphor illustrates our conception of system design, as we described it initially (see section 2. "Research Methodology" on page 3). In itself, the pyramid is not a very original image. It simply helps the designer to ‘view’ his own system within a theoretical framework, for instance to think of the relationship between two corresponding system operators at different levels as the relationship between two schemas in Case’s theory.

3.4. Quality #4: The learner must interact with agents

Learning does not only result from solving problems or using tools, but also from interacting with agents about these on-going activities. Even if the environment is rich and structured, some learners may need help while they solve problems. Some learners need to be pushed forward, otherwise they repeatedly use the same inefficient methods. An agent can of course be human, e.g. a tutor who monitors the learner's work or some peer learner sharing a desk. Regarding computational agents, we distinguish two types of agents:

- ***Agents that conduct domain specific interactions*** (i.e. help to build an experiment). These agents can be domain experts or a computerized collaborative learner (Dillenbourg & Self, 1992b). One can also have several experts that solve the same problem with different viewpoints (Moyse & Elsom-Cook, 1992). The interaction between the expert and the human learner concerns the problem to be solved.
- ***Agents that select and monitor learning activities*** or help the learner to manage his own learning process. In MEMOLAB, we discriminated two kinds of pedagogical agents, the tutors and the coach who is a kind of "tutor in chief". The various tutors have the same function but fulfill it differently according to their style.

This discrimination between two classes of agents has been essential in separating pedagogical knowledge from domain-specific knowledge and therefore to extract from MEMOLAB the components that constitute the toolbox. This architecture will be described in the next section.

4. Agents in ETOILE

4.1. Role distribution

We can summarize the previous section by saying that a learning environment is structured along two dimensions: A sequence of microworlds (with the described features) and a set of agents who interact with the learner within each microworld. By specifying several agents with various roles, we have been able to ***separate pedagogical knowledge from domain specific knowledge***. The separation idea itself had two reasons:

- first, we aimed to build a domain-independent toolbox, i.e. to apply the same pedagogical knowledge on various contents;
- second, we wanted ETOILE to integrate various teaching strategies, i.e. to apply various teaching styles to the same content.

We should point out that the terms ‘domain-independent’ and ‘pedagogical’ are not synonymous. There exists some domain-specific pedagogical knowledge, for instance knowing that problem-23 is more difficult than problem-24. Therefore, we have to cope with the fact that the boundary between what is domain specific and what is pedagogical is not a straight line.

The originality of ETOILE with respect to other AI-based authoring shells such as DOCENT (Winne and Kramer, 1988), IDE (Russell, Moran and Jordan, 1988), ISD (Merrill and Li, 1989) or CML (Jones and Wipond, 1991) is the approach chosen for separating pedagogical knowledge from domain-specific expertise. The common solution to this problem is to create a terminology for describing any subject domain (by using abstractions such as units, chunks, concepts, rules, principles,...). Variables in common pedagogical rules refer to this kind of terminology. The author’s task is to describe the domain using those building blocks. However, building a terminology that fits such diverse domains as astronomy, music and engineering is a challenge that educators and even philosophers have been facing for 2000 years. Therefore, we advocate another approach. The pedagogical rules never refer to the domain. ***The pedagogical rules refer to the interaction between the learner and a domain-specific agent.*** These interactions are described in terms of agreement, disagreement, repair, and so forth. The domain-specific agent is (in the current implementation) an expert system able to solve any problem submitted to the learner. The domain specific terminology used by this expert is unknown to the tutor, it is confined to the expert (although shared with the learner) and therefore escapes from generality constraints. We will describe how the tutor influences the expert-learner interactions, but in order to do that we must first explain how the learner and the expert collaborate.

4.2. How to implement the interaction between the learner and a computational agent?

Imagine two expert systems thar collaborate closely. Each expert has its own rule base, but they share the same facts representing the problem state. Any action performed by one expert changes the set of facts and is thus noticed by the other expert. Thereby, they permanently have a common representation of the current

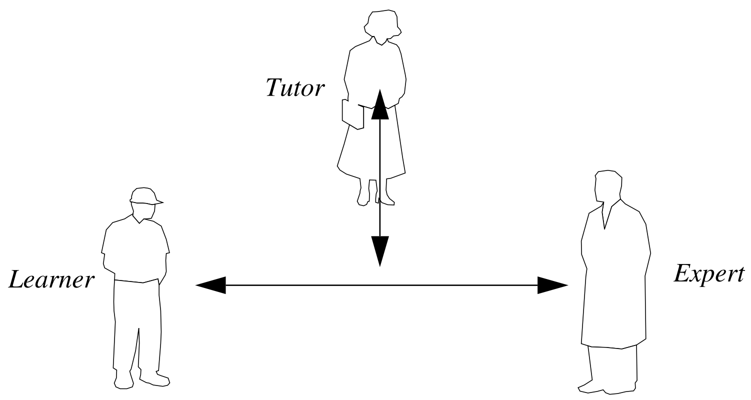

Figure 4: The learner-tutor-expert triangle in ETOILE: the tutor monitors the interaction between the learner and a domain-specific agent.

problem. However, our challenge was to get the expert system to collaborate with a human user, not with another expert system. Nevertheless, we applied the same principle of “sharing” facts: instead of an internal representation, based on predicates, the expert uses an external representation, based on interface objects. If the learner changes the problem state, the expert’s representation is updated. Conversely, if the expert changes the problem state the learner can see the change.

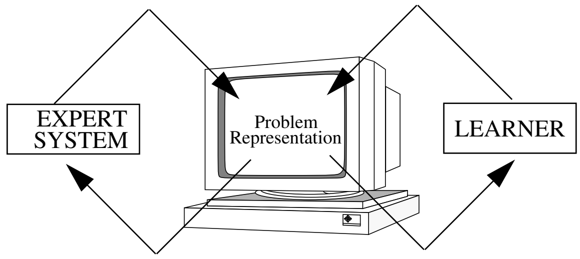

Figure 5: Step-by-step collaboration between the learner and the expert

The main implication for the expert’s rule base is that most rule conditions refer to the problem state displayed on the screen and most rule conclusions change this problem-state display. The relationship between the expert rules and screen objects has been possible because of the object-orientedness of our inference engine (see Appendix III: “The Inference Engine” (page 28)).

The need for shared workspace appeared during our experiments with a previous collaborative system (Dillenbourg & Self, 1992b). In this system (PEOPLE POWER), human-machine collaboration was based on a dialogue about the task (instead on shared task-level actions). Within a shared workspace, if one partner wants to help the other, he can see the other’s partial solution and provide concrete help by performing the next operation (Terveen et al, 1991).

The collaboration between the expert and the learner is performed step-by-step. A machine step is a sequence of rule firings that ends when an activated rule executes a ‘main command’. A learner step is a sequence of interface actions that ends with a ‘main command’. Consequently, a main command is a command that can be performed by both the learner and the expert. It serves as a basis for comparing the learner’s actions with the expert’s potential actions (see section 4.3. "Diagnosis or Agreement" on page 11). At each step, the expert may either ‘***propose***’ an action or ‘***observe***’ the learner’s action. It is generally the tutor who decides whether the next step should be performed by the expert or by the learner. In our terminology, the tutor decides the ‘***local interaction mode***’ (LIM; ‘local’ meaning ‘a single step in the solution path’).

The global balance of learner’s and expert’s steps along a problem solving path determines the degree of collaboration between the learner and the expert. We refer to it as the ***global interaction mode*** (GIM). Its state is represented by an integer between 0 and 5. A global interaction mode equal to 0 corresponds to a ***demonstration*** done by the expert (the expert performing all the steps). A global interaction mode equal to 5 corresponds to a simple ***exercise*** that the learner performs alone. As it is implemented, the interaction mode is based on a quantitative criterion, i.e. the number of steps performed by the learner and the expert. This is of course a limitation of our system because during problem solving some steps may be more critical than others (Dillenbourg, to appear) and it represents an interesting issue for future research.

Let us make clear that collaboration in the system remains flexible. The learner may always interrupt the expert in order to continue the task by himself. Conversely, when the learner is asked by the tutor to perform several successive steps, he is allowed to ask the expert to take over the task. The tutor may disagree with these learner’s initiatives and coerce the learner to proceed with the same interaction mode. However, in case of repeated disagreement, the tutor will resign and ask the coach to select another tutor, more adapted to the learner’s requests (see the notion of pedagogical drift in section 4.6. "Selecting teaching styles" on page 14).

The expert has no pedagogical intention. Of course, the design of the expert is biased by pedagogical criteria. They can be found for instance in the granularity of rules or in the explanatory power of the comments associated with each rule. Nevertheless, during interaction, the expert’s only goal is to solve the problem. It tries to maintain the minimal level of mutual understanding necessary to the joint accomplishment of this task. The expert does not try to teach, or to decide what is good for the learner. He asks questions to get

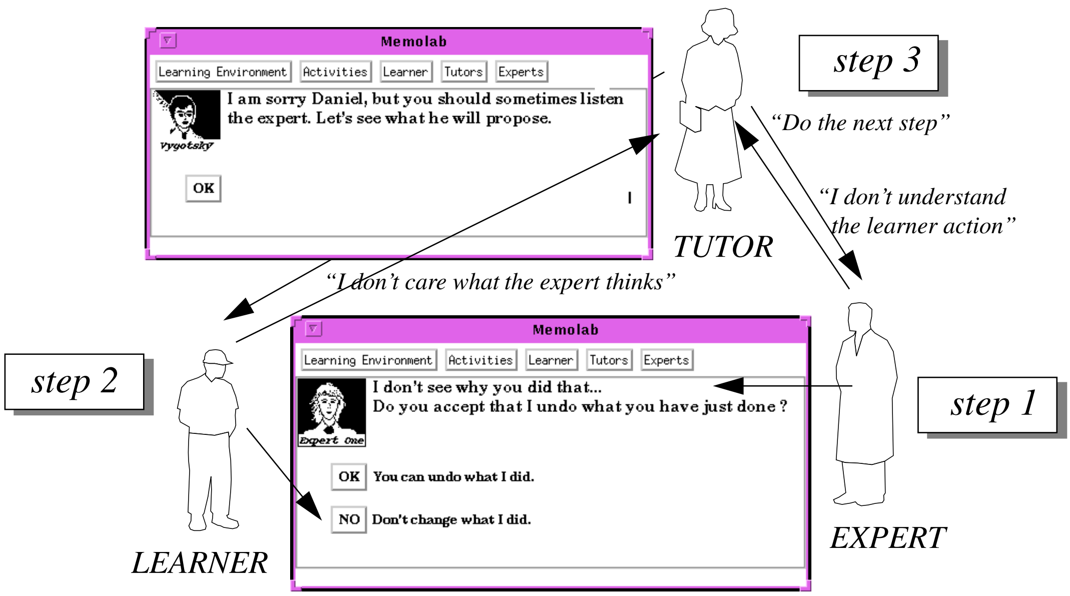

Figure 6: The tutor solves a Nth conflict between the learner and the expert.

answers, not to see if the learner knows the answers. Pedagogical decisions are the tutor's business.

4.3. Diagnosis or Agreement

The role of an expert in a learning environment is not only to guide or interact with the learner, but to detect the learner's mistakes and eventually to understand the cause of these mistakes in terms of missing or erroneous knowledge. This diagnosis process is often referred to as 'learner modelling'. Diagnosis in ETOILE is inspired by the model tracing paradigm (Anderson, 1984). However, within the more collaborative style of ETOILE, we should rather use the word 'agreement' instead of 'learner modelling'. We consider three types of diagnosis:

- 'Positive' diagnosis
  The expert agrees with the learner if one of the rules that he could have activated leads to the same interface command as the command performed by the learner (same command and same arguments).
- 'Unknown' diagnosis
  If none of the rules that the expert was ready to activate corresponds to the learner's action, the expert states that he does not understand the learner.
- 'Negative' diagnosis
  If, after the learner's action, the expert triggers a 'repair rule' (explained below), this means that the learner's previous action produced something wrong, something to be repaired.

The expert has a small set of 'repair rules'. The concept of repair rule is close to the concept of malrule that has been intensively used in student modelling (Dillenbourg & Self, 1992a). The expert's normal rulebase implements a preferred optimal solution path for a task. However, interaction with the learner may move the expert away from optimal to other possible solutions. Outside the solution path, we encounter erroneous problem states that are not covered by the expert's normal rules. Hence, we added a few 'repair' rules enabling the expert to recover from these situations. Determining these situations a priori implies to anticipate the classical learner mistakes. For instance, in MEMOLAB, the expert can repair a list of words which is unevenly distributed (all the long words at the beginning and the short one at the end). This diagnosis technique raised a set of technical problems that led us to modify some aspects of the inference engine (see appendix section III.3. "Adapting OOPS to ETOILE" (page 31)).

4.4. The learner-expert-tutor triangle

The tutor monitors the expert-learner interactions: he receives messages from the learner or from the tutor about what is going on. On the basis of this information, the tutor takes a pedagogical decision. For instance, the expert may send a comment that the learner is wrong, and the tutor will decide whether, in case of error,

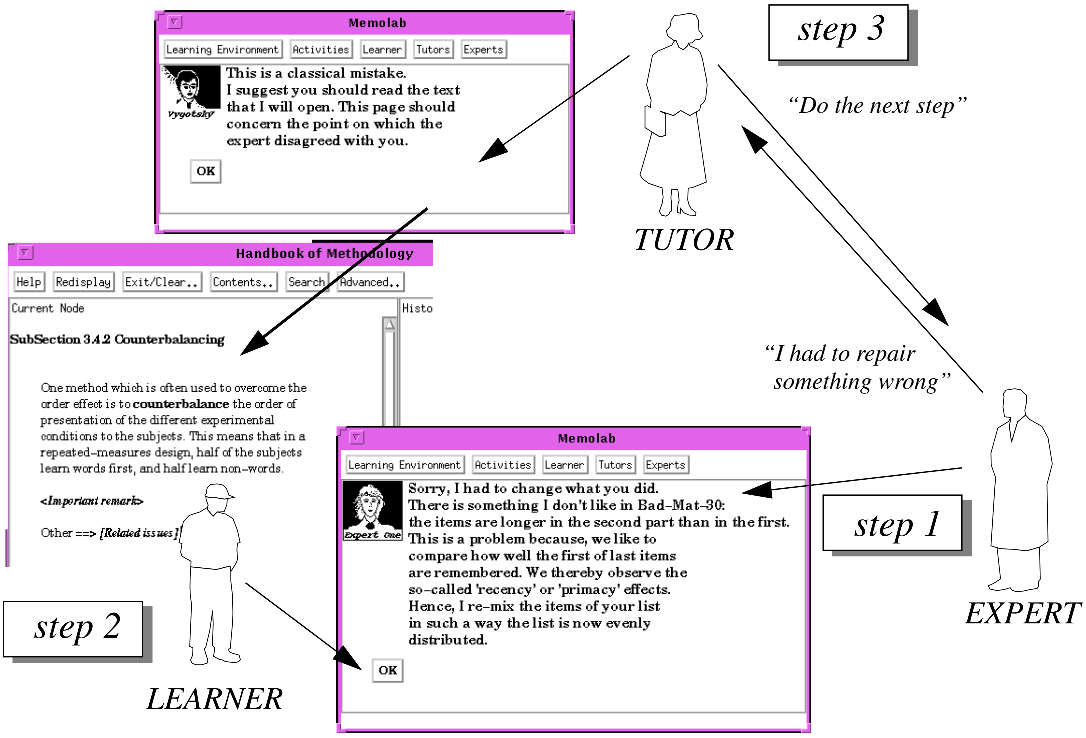

Figure 7: The tutor opens the hypertext because the expert told him that he had to 'repair' what the learner did.

the learner should be interrupted or not. This decision depends on the tutor's teaching style: one tutor could decide an intervention while the other would not care. We illustrate this triangular relationship by two short scenarios. Messages from and to the learner are materialized as popup windows, messages between the expert and the tutor are not visible on the screen

In the first scenario (Figure 6), the learner performs some action not understood by the expert ('unknown' diagnosis). The expert then asks the learner whether he can 'undo' what the learner did. He also sends a message to the tutor to inform him that he did not understand the learner (step 1). The learner has the possibility to refuse that the expert undoes his action. He answers "no" to the expert, this refusal being perceived by the tutor (step 2). The tutor then has to solve a conflict between the learner and the expert. In Figure 6, the tutor enforces the expert's initiative: he tells the learner "listen to the expert" and he asks the expert to continue (step 3). In order to solve this expert-learner conflict, the tutor does not try to determine who is right. He could not take such a decision since he hasn't any domain knowledge. By definition the expert is right. However, the tutor may decide for pedagogical reasons that despite the learner' mistake, it may be better to leave him exploring his own ideas. The tutor illustrated in Figure 6 (called Vygotsky) actually hands control to the learner at the first conflict, but gives preference to the expert at the second one (for the same problem). The dialogue illustrated in Figure 6 corresponds to the second time the learner refuses to listen the expert.

Figure 7 illustrates another mini-scenario. The expert fires a repair rule and informs the learner about the reasons why he changed something on the screen (step 1). The expert sends another message to the tutor to say that he had to repair something that was wrong. After the learner has read the expert's message (step 2) and has clicked on 'ok', the tutor processes the expert's message. In the dialogue illustrated by Figure 7, the tutor's decision is to provide the learner with some theory related to the learner's mistake. The expert's repair rules which correspond to classical mistakes, are connected to specific nodes of the hypertext. Another tutor could decide not to open this hypertext and to leave the learner trying to discover these facts by himself.

This last scenario illustrates the debate about the separation of pedagogical and domain knowledge. The expert is an egocentric agent: he 'repairs' the experiment because he wants to build a good experiment, not because he wants to teach the learner. The pedagogical decision is taken by the tutor. However, expert design is not completely neutral from a pedagogical viewpoint since the anticipation of the typical learner's major mistakes are encoded in the rules.

4.5. Teaching styles

The tutor has to take several pedagogical decisions, some of which already have been described:

- to choose the value of the global interaction mode;
- to choose the value of the local interaction mode;
- to solve conflicts between the learner and the expert ('undo');
- to choose what to do in a case of a 'repair' by the expert;
- to choose the next problem to be proposed to the learner and the expert;
- to decide when to present theory (from the hypertext), i.e. to prefer an inductive (examples first, theory next) or deductive (theory first, examples next) approach;
- to decide when to end the session concerning a specific goal (success or failure);
- to decide to resign before the end of a goal session (explained in section 4.6. on page 14).

Each of these decisions implicitly define an axis along which tutoring behaviour can vary. Originally, in our system, tutor behaviour was summarized by a set of parameters, corresponding more or less to these axes. The value of each parameter was chosen by a set of rules. When we wrote these rules, we learned that the parameters were actually not independent from each other. It is for instance difficult to marry a inductive approach with a high level of interruptiveness. ***The concept of teaching style stems precisely from the consistency among a set of pedagogical decisions***.

With all those teaching parameters it is theoretically possible to define a very large number of teaching styles. However, we reduced this large space to a few styles in order to define styles that differ significantly from each other. There are two reasons for working with a few ***contrasted teaching styles***. The first reason is that if the learner fails to reach a goal with a tutor applying some style, there is a low probability that he would succeed with an almost identical style. The system does better to choose a radically different approach. The second reason for having clear differences between teaching styles is that the learner should perceive these differences (Elsom-Cook, 1991), namely be able to choose his 'favourite' tutor later on.

The five teaching styles we have defined are respectively labelled Skinner, Bloom, Vygotsky, Piaget and Papert. These names do not imply that the we succeeded in translating the theories of these five key figures in the history of education and psychology. We rather use these terms as flags: we defined a tutor that roughly teaches 'à la Vygotsky', another one 'à la Papert' and so forth. Each teaching style corresponds to a tutor and is implemented as an independent rule base. Each rulebase is itself divided in three rule sets: (1) selection of activities and problems, (2) monitoring of joint problem solving and (3) ending a session. There is of course some overlap between the knowledge of each tutor, but it is easier to manage them as separate rule bases.

The only tutor that has so far by-passed the stage of an early implementation and goes through some experimentation is Vygotsky. Before investing in the full implementation of other tutors, we need to solve the problems related to the expert-learner interaction (see section 7. "Experiments" on page 18). From the outset, Vygotsky selects rather difficult problems. The expert initially performs most of the problem solving steps (GIM = 2). Then Vygotsky progressively increases the global interaction mode, i.e. the expert fades out and the learner plays an increasingly important role in the solution process, until he solves the problem alone (GIM = 5). In other words, the Vygotsky tutor is characterized by a particular GIM curve (GIM X Time), while Skinner would be depicted by a 'problem difficulty' curve.

We must acknowledge that the design of teaching styles is bounded by the general architecture of ETOILE. The various components of ETOILE already have some pedagogical bias: the coach's goal selection process is inspired by mastery learning, the expert-learner interaction is inspired by apprenticeship, and the environment facilities are inspired by the microworld philosophy. If we had wanted to push to its end the logic of each teaching style, ETOILE would become much more complex. For instance, the notion of diagnosis is not the same (a) in a behaviourist feedback based on the learner's performance, (b) in a constructivist approach based on the learner's cognitive structure, and (c) in the apprenticeship mode where the expert attempts to integrate the learner's action into his own framework.

4.6. Selecting teaching styles

Since ETOILE includes several tutors, it needs a hierarchically superior agent which selects the tutors. His name is ‘the coach’. The coach also selects the goals in the pedagogical curriculum. The goal selection process and tutor selection process have been implemented in the simplest and most transparent way. The coach selects the first goal whose pre-requisite goals have all been mastered (remember that the curriculum is a network of goals with prerequisite links). When the learner succeeds that goal, its status is set to ‘mastered’ and the goal selection process is repeated.

After a goal has been selected, the coach selects a tutor. The mission of a tutor is always relative to a single goal. Goal selection is mainly based on the learner’s preferences and his performance history. If he has already failed the current goal with tutor T1, the coach will select a tutor T2 that provides more guidance than T1. On the contrary, if the learner succeeded the previous goal with tutor T1, the coach will select a tutor that provides less guidance. Our five tutors are sorted by decreasing level of directiveness: Skinner works step by step, Bloom makes larger steps but with close control of mastery, Vygotsky is based on participation, Piaget intervenes only to point out some problems and Papert does not interrupt the learner.

This general principle is however shadowed by more specific rules that recommend to avoid activating a tutor that has been repeatedly inefficient with a particular learner. The learner may also select a tutor. These preferences are recorded and taken into account in later selections. The learner (or a human teacher) can also remove a tutor from ETOILE’s set of tutors for the rest of the session.

The tutor has rules to determine when the learner has mastered or not a goal and he can close a session concerning a goal. In order to provide more flexibility to the system, tutors can resign before the end of their ‘contract’. If a tutor like Piaget observes that the learner is frequently asking for help or for theory, this means that his teaching style does not fit the learner needs. Conversely, if the learner being taught by the tutor ‘Skinner’ always asks to continue the problem on his own, he would certainly be more efficient with another tutor. We refer to this process a ‘***pedagogical drift***’, because the actual interaction with the learner may lead the tutor to ‘drift away’ from the behavior space for which he has been designed for and within which he is supposed to be efficient. In case of drift the tutor returns the control to the coach and indicates the type of drift that he observed. The coach will then select another tutor.

This monitoring of pedagogical drift is implemented in a very simple way: each learner command that has an effect on the locus of learner activities’ control creates as a side-effect an instance of the class ‘initiative’. This class possesses two subclasses ‘want-more-guidance-initiative’ and ‘want-less-guidance-initiative’. Some commands like seeking expert help or asking to read the hypertext generate instances of ‘want-more-guidance-initiative’. Some learner actions, like ‘I want to continue’, ‘I want to select the problem’, ‘I refuse to undo’, create instances of ‘want-less-guidance-initiative’. All significant commands in ETOILE are characterized with respect to these two subclasses. Therefore, all domain-specific commands relevant at this level must create an instance of one of these two sub-classes. The tutor compares the number of instances in each subclass and resigns if there is a strong disequilibrium between the two sets of used commands.

5. Developing a new learning environment with ETOILE

Creating courseware with a traditional authoring tool means preparing a scenario. This scenario is composed of activities, often based on the canonical schema: question - answer - feedback. The author prepares each question, each response analysis process and each the feed-back item. He specifies in a detailed script how these activities link to each other. In ETOILE, the curriculum grains are much larger. The curriculum is a sequence of microworlds plus a set of domain-specific agents.Those agents will interact both with the learner and with the pedagogical agents (tutors) included in the toolbox. The originality of ETOILE is here: ***designing a system is not preparing questions, it is adding new agents*** to those included in the toolbox and creating the context (the microworld) where those agents and the learner interact.

The backbone of ETOILE is the ***curriculum***. All knowledge has to be plugged into this structure. The curriculum is a network of goals, connected by prerequisite links. Each goal is an object with four slots:

- The **problems**. A goal defines a skill by specifying a set of problems that the learner must be able to solve.
- The **interface**. Learners solve problems through a goal-specific interface. This interface is basically one or more windows, with a set of commands (menus, buttons, moves,...).
- The **domain expert**(s). An expert for goal-X is a rulebase able to solve any problem stored in goal-X.
- The **hypertext**. A goal includes a hypertext chapter that the learner may explore freely and where he can find any information relative to the skills he must acquire.

This representation of goals results from our design principles. Each goal has its own interface and expert, this feature supporting the particular architecture of MEMOLAB, especially the language shift principle. However, the author may also choose to use the same expert and/or the same interface for several goals. The designer's task is not automated. The implementation of an ILE with ETOILE follows the following steps:

(1) The developer defines the ***curriculum*** by creating a set of instances of the class 'goals' and by specifying prerequisite relationships among those goals (pointers from one goal to another one).

(2) For each goal, the developer creates a set of instances of the class '***problems***'. He may create a subclass of problems (e.g. "equations") into which he adds the domain-specific information needed by the expert or the simulation. For each problem, the developer specifies its relative difficulty (with respect to the other defined problems) by an integer between 1 and 5.

(3) The developer creates the hierarchy of ***domain-specific objects*** (classes and instances) that will be manipulated by the learner and the expert during problem solving.

(4) For each goal, the developer defines one or more ***application frames***, i.e. windows wherein the learner will solve problems. The application frame is a concept from the Common Lisp Interface Manager (CLIM). It gathers all the information that defines a situation, i.e. the window, the command tables, etc. Learner commands are stored in command tables. Two goals may share the same frame. Some components of a frame may be large: for instance, the simulation used in MEMOLAB, though it is a large piece of code, is only a subcomponent shared by the microworlds.

(5) For each goal, the developer creates an '***expert***', i.e. a rulebase that is able to solve the problems stored in that goal.Within the framework of ETOILE, the conditions of the expert rules must refer to the problem state displayed on the screen and the conclusions must include commands that change something on the problem display (commands that belong to the learner command table). If for the particular domain, the same problem can be solved by different approaches that can not easily be covered by a single rulebase (e.g. multiple viewpoints), the developer should define several experts with respect to a same goal. If several experts are available, the active expert will be selected by the coach with the same criteria he uses for selecting tutors (efficiency and preferences). However, if the expertise of one expert is qualitatively or quantitatively superior to the expertise of another, then these experts should be associated with different goals. We aim to extend this architecture to include other kinds of domain specific agents, such as computerized collaborative learners.

(6) The developer writes one or more ***hypertexts***. For each goal, he specifies a hypertext node to be considered as the entry point for displaying theory related to that goal. The hypertexts must be structured in multiple granularity layers, in such a way that the learner can deepen some points and pass quickly over others. The hypertext should be integrated with the rest of the application wher-

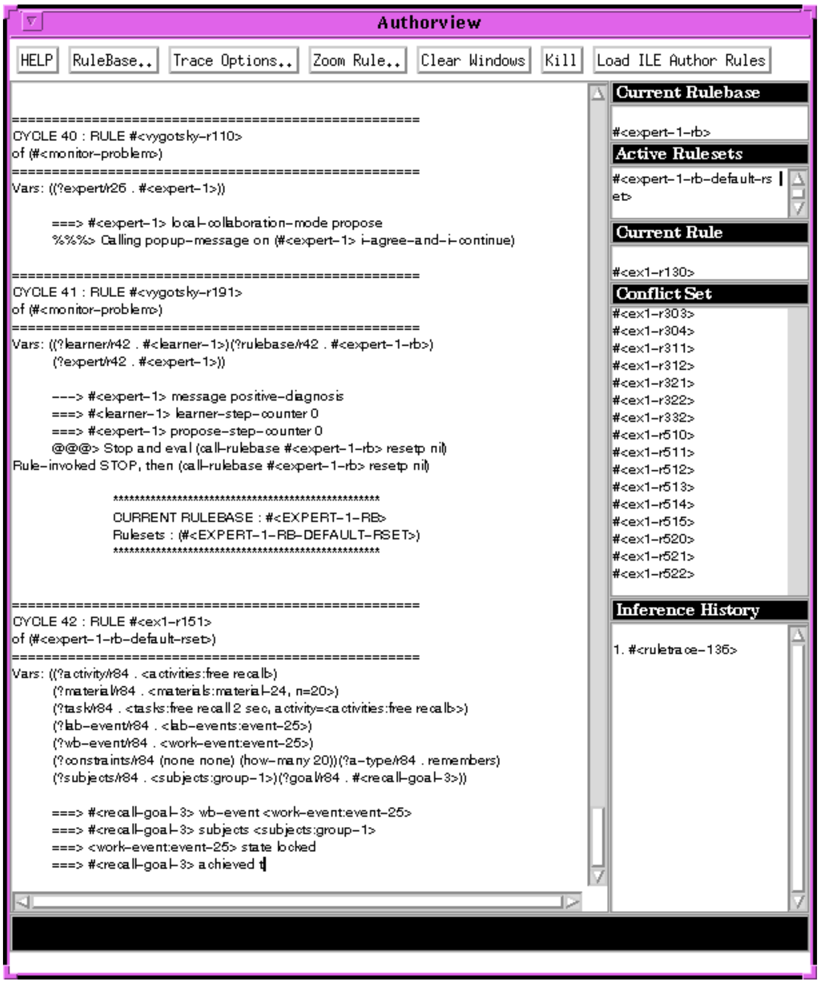

Figure 8: The 'Authorview' window enables the developer to monitor the execution of a rulebase and the activation of the rule bases of various

ever the learner may need information in a hypertext format. This is done by including 'external buttons' (that directly open a hypertext to a specific node) at various places in the application frames (step 4).

(7) The developer anticipates the learner's plausible main mistakes and he prepares 'repair rules' that will enable the expert to correct those mistakes. He associates those repair rules with hypertext nodes in order to provide the learner with some information relevant for his mistake.

In summary, the author's job includes three kinds of tasks: (i) creating subclasses and instances of classes defined in ETOILE, (ii) building the interface, its objects and commands, and (iii) writing rule bases. The first task is easy, the second is completely in the developer's hands, but the third one is more complex and requires a good understanding of our inference engine's workings. Hence, we developed some tools that support the design of expert rule bases.

As we said in the beginning of this report, we did not intend to produce a real authoring interface. We simply added a few functions that enable the author to activate or disactivate an agent (tutor, coach or expert), to reset its rulebase, to test an expert non-interactively (i.e. to check whether an expert is able to solve a problem without any interaction with the learner). We also created an interface that allows the developer to follow step-by-step the execution of a rulebase. This tool is illustrated by Figure 8.

6. Implementation

ETOILE and MEMOLAB are implemented with Allegro Common Lisp and run on Sun Sparcstation. It could be easily ported to other Unix platforms. Since the whole systems are based on an object-oriented approach, we use the object-oriented features of Common Lisp, i.e. CLOS. For the interface functions, we use the beta-release of the 'Common Lisp Interface Manager' (Clim 2.0). The software may be ported to micro-computer platforms (i.e. Windows and MacIntosh) when Clim 2.0 and multi-threading will be available for those in 1994.

The main components of ETOILE are:

- the Object-Oriented Production System (see section Appendix III: "The Inference Engine" on page 28) that includes not only the inference engine itself but also all the classes used by this engine (rules, rule bases,...);
- the hypertext (see section Appendix II: "The Integrated Hypertext" on page 26) which is also implemented as objects and has hence been integrated with the other system components;
- the hierarchy of pedagogical objects: goals, problems, initiatives, and the classes of agents (the coach, the tutors and the experts);
- the instances of pedagogical agents: the coach and the tutors, with their rules;
- the 'Learning Environment' window which contains commands that are supported by ETOILE, whatever the domain is: starting/ending a session, selecting a tutor, an expert or a problem, asking to see the expert's goals or the expert's trace, opening an hypertext...
- the 'Developer Tool' window which contains commands that enable the author to test an agent, to reset its rulebase, etc...
- the 'Authorview' and 'Developer Tools' windows (see section 6) that provide the designer with basic facilities to monitor and control agents and the execution of rulebases.

The components of MEMOLAB are:

- the curriculum, i.e. a list of instances for the classes 'goal' and 'problem';
- the instances of experts, i.e. a rulebase for each expert;
- the instances of hypertext: the "Handbook of Methodology" and the "Encyclopaedia of Memory"
- the domain-specific application frames, i.e. the windows where the lab appears, with all the commands for designing an experiment, the window that helps building a list of words, the simulation window,...
- the domain-specific classes and instances to be manipulated by the learner: experiments, events, tasks, subjects, materials, words, non-words,....;
- the simulation (which is conceptually included in the domain-specific frames)

In other words, MEMOLAB includes two sets of things. The first three are instances of pre-defined ETOILE classes. The author may define sub-classes if he wants to add domain-specific slots to these objects. The three next things define the microworld. The author of a learning environment must add in ETOILE the file name defining the file structure of his own learning environment (using 'defsystem').

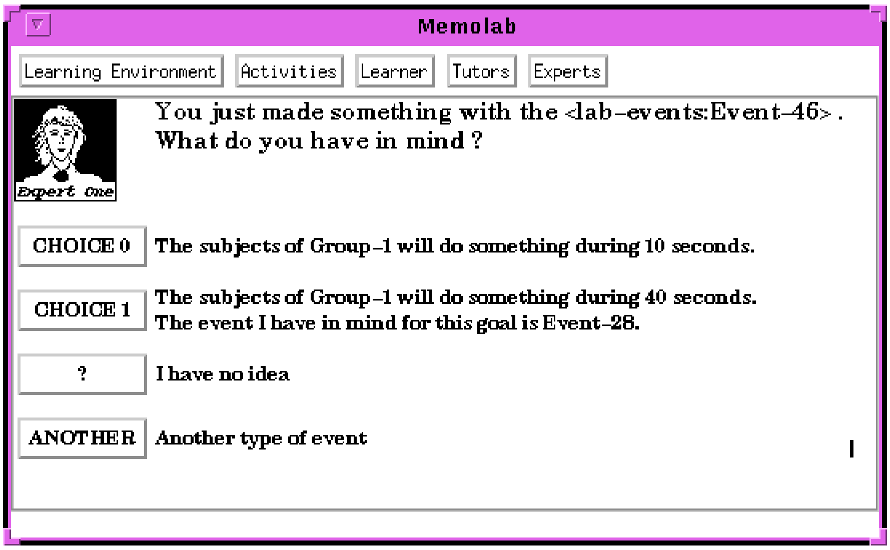

Figure 9: If the expert cannot really attribute the learner actions to one of his own goals, he asks the learner what was his goal.

7. Experiments

We have conducted preliminary experiments with a few subjects. These experiments were too early to draw conclusions regarding the efficiency of MEMOLAB. Before extending experiments to samples supporting statistical inference, we have to incorporate these preliminary results into our systems. Experimentation pointed out several difficulties in the learner-expert interaction and this is important because this interaction is a key factor for role distribution among ETOILE's agents.

When the learner creates an event on MEMOLAB's workbench, the expert tries to map this event to task goals related to his own problem solving activity. In other words, the expert attempts to understand what the learner does. This interaction is illustrated by Figure 9. Bijective linking between objects on the screen and task goals was implemented in order to optimize the expert, but it raises several problems when an event may be attached to several task goals. If the expert has planned that Group-1 will wait for 10 seconds and Group-2 will wait for 40 seconds, the learner may very well have in mind that Group-1 will wait for 40 seconds and Group-2 for 10 seconds. We use the term 'goal commutativity' to express the fact that the allocation of parameters to experimental samples is arbitrary, and that the expert must be able to permute his goals.

However, mapping student actions to task goals raised several issues. Firstly, learners don't necessarily have goals, but instead start to think about goals when they are proposed. This is not a bad point. The second issue concerns the way in which the expert describes his goals to the learner. For instance, in Figure 9, the expert's goal described in 'choice 1' is actually that Group-2 waits for 40 seconds, but it would be nonsense to propose a goal concerning Group-2 when referring to an event concerning Group-1. Hence, the expert generalizes his specific goal in order to cover the learner's intentions. This led to ambiguous situations where the expert was suggesting different goals that were actually expressed by the same sentence, generating a misunderstanding between the learner and the expert.

However, these difficulties gave us the opportunity to observe instances of 'social grounding'. Social grounding is the mechanism through which each participant in a dialogue elaborates the belief that his partner has understood what he said, to a level sufficient for continuing the conversation (Clark & Brennan, 1991). In the protocols we encountered utterances (produced in French and translated by us) showing that the learner monitors the understanding that the system has about himself:

> *"He supposes that I wanted the subjects to do something during 40 seconds. I wanted the subjects to do nothing."*
> *"He does not understand why I did that? "*
> *"Precisely, what I want is that the subjects do nothing, I think that it is what he thinks"*
> *"I am sure he will tell me again that I am wrong"*
> *"He will ask me what I wanted to do. And the..., since... he'll base himself on wrong things since this is not what I want to do."*

In other words, the learner-expert interaction is not simply based on mutual diagnosis with ***nested levels of beliefs***: the expert diagnoses the learner, the learner diagnoses the expert, the learner diagnoses the expert's diagnosis,... From a theoretical viewpoint, one could go on about the diagnosis of the diagnosis of the diagnosis... However, in human-human dialogue, this cascade is not very long because human conversation is very rich in resources to detect (e.g. facial expressions) and repair (e.g. pointing) communication breakdowns. As soon as we reach the level "diagnosis on diagnosis", one speaker can repair the dialogue. In the fictitious example below, speaker 1 repairs the understanding that speaker 2 has about his first utterance.

Speaker 1: *"Numbers ending with 4 or 6 are even numbers"*
Speaker 2: *"778 is also an even number"*
Speaker 1: *"I did not say that those ending with 8 are not even"*

When a speaker refines a previous statement to correct some misunderstanding, quite often he does not simply repeat what he said but reinterprets it from the point of view of his partner. The fact that the learner reinterprets his own actions within the expert's conceptual framework corresponds to a learning mechanism that has been emphasized in the socio-cultural approach: participation changes understanding (Newman, 1989; Rogoff, 1991). The benefits from collaborative work seem to be grounded in the mechanisms engaged to maintain a shared understanding of the problem (Roschelle,1990).

Through these results, the process of cognitive diagnosis appears as mutual and reciprocal. Instead of having one partner who builds a diagnosis of the other, we have two collaborators who try to build a mutual understanding (Bull, Pain & Brna, 1993). A similar evolution can be observed in work on explanation: the explanation is no longer viewed as a structure built by one partner and delivered to the other, but as a new structure jointly constructed by two partners during their interaction (Baker, 1993; Cawsey, 1993). This evolution of diagnosis and explanation techniques drives the evolution of cognitive science towards the idea of distributed cognition (Resnick et al, 1991; Shrage, 1991; Dillenbourg, to appear).

8. Synthesis & Future Developments

This research has produced different types of outcomes both at the design level and at the implementation level. Below, we list both the outcomes of this research project and some avenues for continuing research in this domain:

(1) We developed an original solution for ***separating domain-independent from domain-specific knowledge***. Instead of searching for a domain-independent language able to describe any domain, ***we describe pedagogical knowledge in terms of interactions among agents***. A learning environment is a society of agents for which we have specified (1) the role distribution over agents (coach, tutor, experts) and (2) protocols of communication among agents (global and local interaction mode, repair, conflicts,...). Some of the choices made in ETOILE are still arbitrary. For instance, the only domain-specific agents we implemented are 'experts' while we could integrate 'peer learners' or semi-experts. Another limitations is that 'agents' refer to a simple data-structure, including namely a rulebase that drives his behavior. Current agents do not have the autonomy and goal-orientedness that characterize a proper multi-agent system (Soham, 1993). The specification of well-defined communication protocols among agents would enable the integration of heterogeneous agents, namely agents which do not rely on rule bases or agents written in a different programming language. This approach would also extend ETOILE's scope to designers outside the LISP community.

(2) By separating pedagogical from domain knowledge, we are not only capable of permuting the domain knowledge associated with some teaching strategy, but conversely, we are also able to apply various teaching styles to a particular domain expertise. Instead of arguing which teaching style is the best, we prefer to cover a ***large range of teaching styles***. Of course, this raises the issue of selecting the teaching style. Despite this issue being non-trivial from some theoretical viewpoint, we solved it with a simple pragmatic principle: if the learner fails, change the teaching style! We would like to develop future research on this learner's perception of teaching styles.

(3) A third outcome of our work lies in the design metaphors used for developing MEMOLAB and ETOILE. The pyramid metaphor is in itself very simple, but it enabled us to ***structure our sequence of microworlds on the basis of some theory of cognitive development***. This metaphor is closely connected to the ***language shift principle*** which helps developers to smooth the transition between two microworlds using different interaction languages. In the near future, we would like to collect more empirical data to check the theory's validity and its application to MEMOLAB, and to observe the learner adaptation mechanisms when a language shift occurs.

(4) Another achievement concerns the integration of multiple tools within a learning environment, especially the ***integration of a hypertext with rulebase agents***. Nowadays, hypertext systems are a very popular research topic. We believe that the pedagogical power of a hypertext cannot be understood without considering the context that justifies information search in a hypertext.

(5) A fifth lesson from this work concerns the interaction between the leaner and a computerized agent. The principle we applied is very simple: ***human-computer interaction should concern what is ON the screen***. The design of agents, especially of the expert, has been determined by this principle: the rule conditions 'read' the problem state on the screen, the rule conclusions change the problem state display. This design principle produces a nice opportunistic expert-learner collaboration and makes this interaction inspectable and tunable by the tutor. Experimentation with users helped us to find problems related to this approach, namely the difficulty of interacting about goals. But they also gave us some insight for implementing mechanisms of ***social grounding*** between a human user and a machine. Social grounding is a mechanism of joint and reciprocal modelling between two partners, based on the negotiation of shared meanings. A limitation in our current architecture is that the rule variables are instantiated by screen objects without any ambiguity. A new research avenue would test interactions where the computer's concepts were not directly connected to the displayed objects (the learner concepts are neither), and where these mappings could be negotiated with the learner through various pointing mechanisms.

Appendix I:
The Simulation

The simulation component is called the Memory Machine (MM). It belongs to the domain specific part of MEMOLAB. However some solutions used for implementing this simulation might be useful for building simulations in similar domains, more precisely in domains lacking mathematical models to compute results.

The input of the simulation is the experiment that the learner has built in the LAB. In an experiment, pseudo-subjects are supposed to memorize words or other items and later try to remember them. The output of the simulation is the list of words that each subject has remembered. The constraints that justify our design choices are related to the pedagogical function of MM and to the nature of available knowledge.

I.1. Pedagogical constraints.

The simulation has to produce results as similar as possible to the results obtained in an experiment using real subjects. However, psychological experiments have *de natura* a low fidelity. Moreover, this simulation is clearly not a substitute for real experimentation. Therefore, the validity of simulated results must be assessed in the light of pedagogical goals: to acquire basic skills in methodology of experimentation. This goal determines the lower limit for the validity of MM results. This lower limit can be expressed by the following rule:

> Minimal Validity Rule: if the learner builds an experimental plan with a factor F and based on a paradigm P, and if the literature includes knowledge on the effects of factor F, within the paradigm P, then the simulation must produce data in which the effects F can be identified.

More than being precise, the results must be explainable. Several ILE designers (White and Frederiksen, 1988; Roschelle, 1988) have shown that a qualitative simulation, although less efficient, is better suited to pedagogical purposes. Simulating an experiment involves an analysis of the structure and content of this experiment. The simulation produces a trace that can be used in explanation. It consists of pre-stored comments and keywords. We are exploring the possibility for MM to generate a trace structured in hypertext form. This would allow the learner to read the explanation at the level of detail that he wants.

I.2. Knowledge Acquisition

Contrary to many physical devices, human mnemonic behavior cannot be predicted by a set of formulas or rules. Available theories do not constitute a consistent and exhaustive body of knowledge. Knowledge is distributed among a large set of experiments. Therefore, learners' experiments will be simulated by comparison to similar experiments from the literature. This process is implemented with cased-based reasoning (CBR) techniques (Riesbeck and Schank, 1989).

Moreover, the literature does not cover the very large space of experiments that can be designed in MEMOLAB. Within the space of possible experiments, psychologists have concentrated their work on some avenues, or paradigms. (In experimental psychology, a paradigm is defined as a class of experiments.)The comparison of experiments within a paradigm brings more information than comparing experiments which differ in too many ways. Therefore, the case-retrieval process is not a simple search through a flat set of cases, with some similarity metrics. The set of cases (literature experiments) is partitioned into paradigms and a main stage during the simulation consists in identifying the relevant paradigm (see below, the paradigmatic analysis).

Another peculiarity of this domain is that the adaptation of the retrieved case to the target case cannot be driven by universal rules. Let us imagine that the learner builds an experiment where subjects have to memorize a long list of words and that MM retrieves an experiment which is similar on all points but the list length. The effect of list length is not universal, but varies according to other factors such as the delay between the memorizing and recall events. Therefore, case-adaptation is performed by a set of rules (named Vertical Adapters) that differ for each paradigm. In short, ***case-adaptation in MM is governed by rules that are themselves case-dependent***.

I.3. The Case Library

The library includes the paradigm discrimination tree (Figure 10) and a set of experimental sequences (cases). Each leaf of the paradigm tree corresponds to a subset of sequences found in the literature. After identifying to which leaf node corresponds the learner's sequence, we retrieve the experiment most similar to the learner's one. The vertical adapters (VAs) are stored at different levels of the tree, according to their generality.

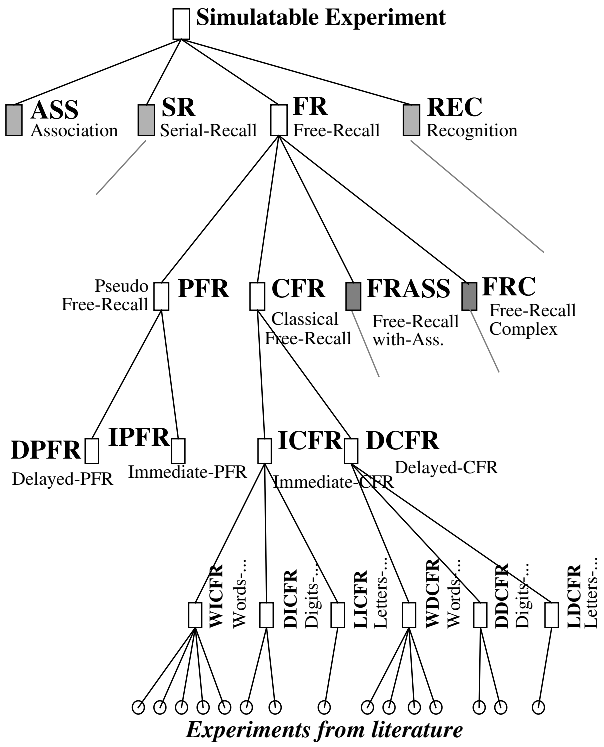

Figure 10: The paradigm discrimination tree or structure of cases in the simulation

I.4. The Algorithm

The simulation considers both the content and the structure of experiments. The content of the experiment refers to the sequences composing the experiment: what are the tasks, the material features, etc. This information is necessary to determine the paradigm to which the learner's experiment belongs. The structure of an experiment refers to the relationship between sequences. The fact that, for instance, two experimental sequences are identical on all points but the material length indicates that the learner intends to observe the effect of the material length. This 'factor' must be identified in order to respect the 'minimal validity rule' (see above).

The ***structure analysis*** returns the factors identified within the learner's plan and the value (modality) of this factor in each sequence. This knowledge feeds the ***paradigmatic analysis*** process which identifies the most specific paradigm (leaf node) corresponding to the learner's sequence. (The word 'paradigmatic' refers here to the meaning used in experimental psychology, as specified above.)

Let us consider first the simplest simulation process, i. e. computing the results for a sequence s_i. This sequence s_i is matched against the discrimination rule of each node N of the paradigm tree. If the match is positive, we store the vertical adapters of N and recurse on descending nodes. When a leaf node is reached, the associated set of literature sequences $S'_i = \{s'_{i.1}, s'_{i.2}, \ldots, s'_{i.n}\}$ is retrieved. In addition, vertical adaptation rules have been collected through the tree: $VA_i = \{va_{i.1}, va_{i.2}, \ldots, va_{i.m}\}$. The selection process reasons on VA_i in order to select among S'_i the experimental sequence s'_i which is the most similar to s_i. The last stage is to retrieve the data of s'_i - denoted D'_i - stored with s'_i, and to execute VA in order to compute D_i:

VA (D'$_i$) ===> D$_i$

Let us imagine that the learner's experiment includes two sequences s_1 and s_2. We can apply independently the same algorithm on each sequence and generate results for each sequence:

VA (D'$_1$) ===> D$_1$
VA (D'$_2$) ===> D$_2$

However, this independent processing does not respect the minimal validity rule expressed above. Let us assume that between s_1 and s_2, the learner has created some systematic differences in order to observe the effect e of factor f (denoted e_f). With independent sequence processing, we cannot guarantee that, when learners will compare D_1 and D_2, they will find e_f. In order to guarantee that differences between D_1 and D_2 correspond to the expected effect, one must generate D_2 from D_1 and what we know about e_f. When s_1 and s_2 are compared to a tree node, the effects stored with that node are compared to the structure of (s_1,s_2). This structure was determined during the structure analysis stage. If e_f is identified, the Horizontal Adapters *(HAs)* that are associated with e_f are stored. Then, the experiment is 'reduced', i. e. only s_1 is passed to lower nodes. Data for s_2 will be generated from D_1 and the VA of e_f.

```
VA(D'1)  ===> D1
HA(D1)  ===> D2
```

These two strategies are summarized in Figure 11. The optimal strategy is applied when a known effect is identified within the learner's experiment and if this effect does not interact with other factors. The optimal solution can be applied on parts of the learner's experiments.

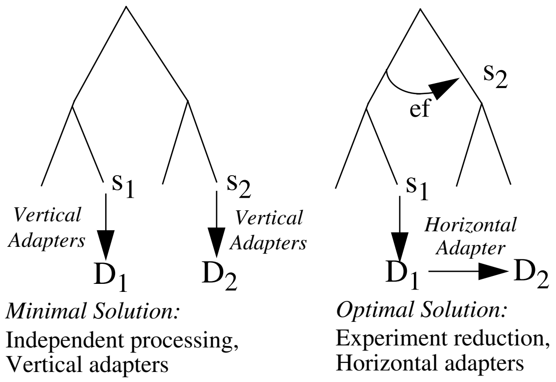

Figure 11: Two methods of computing the performance of each group of subjects

I.5. Output from the simulation

The simulation provides the learner with a window illustrated by Figure 12. This window includes three panes (the last one being shared for graphics or data):

- The simulation ***trace*** shows which experimental data (from the literature) have been used and how data did affect the results. The scientific literature references quoted in this pane appear in bold characters because they constitute ***external buttons of an hypertext***: if the learner clicks on one of these buttons, the system directly opens the hypertext 'Encyclopedia of Memory' at the relevant node'.
- The analysis of variance performs the ***statistical treatment*** necessary for data interpretation. The learner can ask to focus the statistical processing on the first five items of on the last five of the list in order to measure more specifically primacy or recency effects.
- The ***graphics*** pane shows the frequency of correct remembering for each item of the material. The performance of each group of subjects corresponds to a different curve. The learner can modify the scale of this graph.
- The ***raw data*** pane shows the ordered list of words hypothetically produced by each subject of each group. It helps giving the simulation a more "realistic feeling", because the learner can perceive the very "soft" nature of psychological theory by observing important variations among subjects.

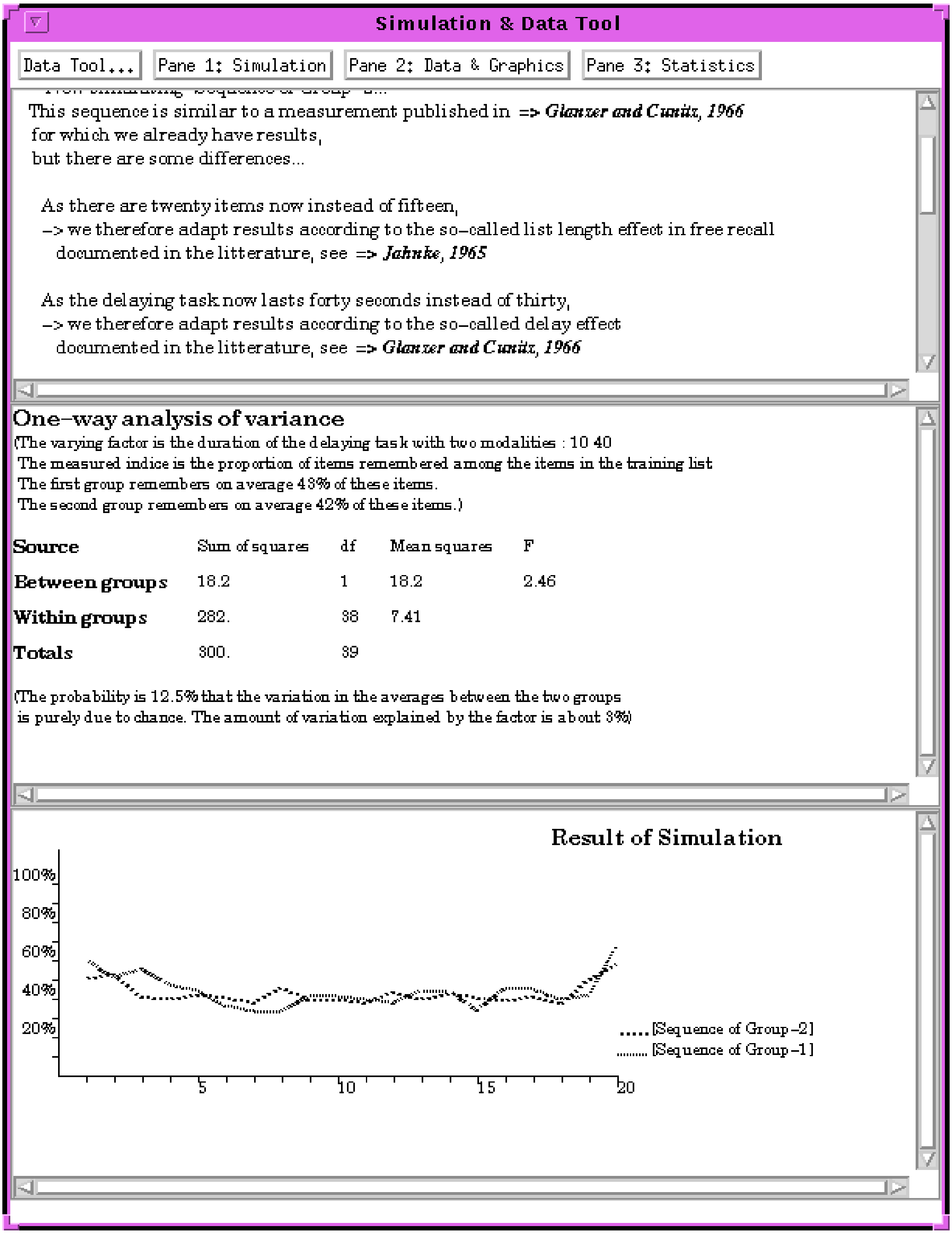

Figure 12: Three of the four panes of the window presenting the output of the simulation. (The graphics pane can be shifted with the ‘row data’ pane). The bold names in the simulation trace are external hypertext buttons.

The third and fourth pane cannot be displayed simultaneously: the learner can shift between these panes by a menu of the simulation window.

In the current version of Memolab, the simulation tool includes the data tools and statistical tools that we first designed as independent tools. We integrated them into the simulation to simplify the environment (which had then too many windows). Our original project intended to have some tutoring related to the analysis of experimental output, both for the statistical processing and for interpreting results with respect to hypotheses. MEMOLAB would then have covered the whole reasoning cycle of an experimental psychologist. However, the focus placed on ETOILE led us to reduce the amount of energy invested in MEMOLAB and to concentrate on the methodology of experimentation.

Appendix II:
The Integrated Hypertext

A hypertext system was deemed the most appropriate computational infrastructure to present knowledge resources offered to the learner. In traditional text processing systems, a document is no more than a sequence of character strings which are stored and retrieved in a strictly linear fashion. The essential innovation of hypertext consists in abandoning this linearity constraint. The hypertext approach views a document as a set of text chunks corresponding to conceptual nodes in a database; a system of machine-supported links between nodes and chunks allows interactive branching within and between documents. Hypertext systems exploit more recent computer interfacing technology, particularly window management and graphics tools. Conceptual entities are displayed as windows of text which may contain any number of icons or buttons representing links to other nodes. The user can navigate freely around the hyperdocument by clicking on any of these graphics pointers.

II.1. Design

While commercial hypertext shells are available, we have implemented our own hypertext system for several reasons:

- Building the hypertexts on a commercial shell would lead to portability and copyright problems for the distribution of our system.
- An intelligent learning environment should give an overall impression of implementational coherence; by building our own hypertext tool, we can freely tailor it to share the look-and-feel of the other system components.
- Commercially available systems are usually closed products, and embedding them in a larger system leads to inevitable communication problems.

The hypertext we developed does not provide new facilities that do not exist on similar systems, its originality is to have a degree of integration with the other system components:

- '***External buttons***' can be placed by the designer in any text pane within any window of the learning environment. When the learner clicks on such a button, the relevant hypertext is open at the specific node.
- ***Rule based agent***s (or any other program segment) can open the hypertext at a specific node. The choice of the node is based on connections between particular rules ('repair rules') and hypertext nodes.
- A program can be run from the hypertext: '***procedural buttons***' trigger any procedure built by the designer, for instance some demo or animation illustrating a concept from the hypertext.

The underlying position did not only focus on how learners seek for information but also on why they need information. More than a simple technical issue, the ***degree of integration measures the extent to which the need for information comes from other activities performed within the learning environment***.

II.2. Implementation

The hypertext subsystem is implemented as a class of objects called hypernodes. Each hypernode contains a title and a chunk of text with embedded formatting commands. Moreover, each node has an associated list of keywords summarizing the semantic content of its main text. These keywords serve as the basis of an indexing facility.

A hypertext is written by using a small subset of Latex commands which then have to be compiled in order to be used on-line. We included book structure commands (chapter, section,...). This way an author can combine the advantages of linear and 'web' text. A special graphic browser allows to display the linear structure of a hypertext and to jump to a section. Because of the use of a Latex, hypertext documents can be printed easily. For instance, the user manual of Memolab is available both as an on-line hypertext or as a printed document.

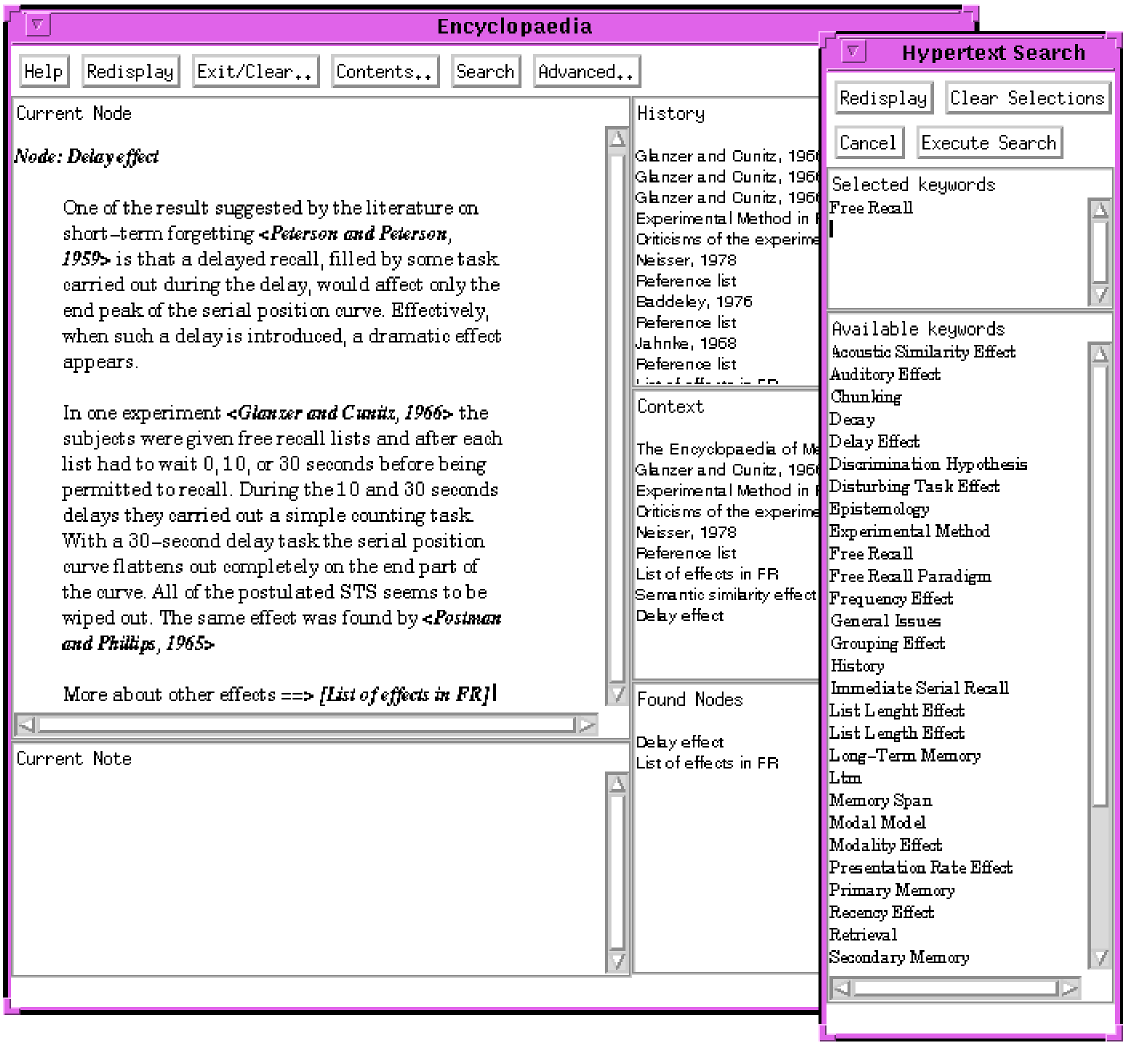

Figure 13:A hypertext from MEMOLAB and the 'search' facility.

The following facilities for 'navigation' have been implemented:

- ***Nodes and notes***
  A node is usually a large piece of text (between 1/2 and 2 pages of text), hence this text is displayed in the main pane. A note is a short piece of text that can displayed in the note pane, at the bottom of the window (see Figure 13). Typically, it provides an information that the learner could like to read briefly, without hiding the main node being read.
- ***Web view***
  The learner may ask to see the structure of the hypertext displayed as an horizontal tree (menu item 'book content'). The learner may ask to view only the portion of the tree locating the node being read within the overall structure (menu items 'book context').
- ***Search***
  The hypertext includes a keyword based search facility: the learner asks for 'search' and selects one or more keywords in a special popup window. The system then retrieves all the nodes indexed by these keywords and displays the headers of the retrieved nodes in the 'found notes' pane. The learner may click on the node he wants to read. These keywords are defined by the hypertext author.
- ***Traces***
  The three right-hand panes provide various common navigation helps: the 'history' pane displays the list of nodes read by the learner, the 'context' pane displays the stack of nodes between the node being displayed and the root node, the 'found nodes' pane displays the list of nodes resulting from a particular search. The learner may click on any line and jump directly to the related node.

Appendix III:
The Inference Engine

The expertise of any agent (domain-specific or pedagogical) is expressed by a set of rules. These rules are executed by the inference engine. We have implemented an inference engine in order to achieve a high level of integration between the rules and the other components of the system. We first describe the generic engine. i.e. an application-independent engine that could be applied to any AI problem. Then, we explain how we added some facilities or modified some aspects of the engine to adapt this generic engine to the architecture of ETOILE.

III.1. The OOPS system

The implemented production system has the following features:

• ***tight integration of object-oriented and rule-based programming***

In first-generation rule based systems, the inference engine was assigned a special working memory, neatly separated from other data structures such as those manipulated by conventional code. This working memory was represented as a list of assertions which were matched with rule conditions to determine applicable rules. The advent of object-oriented languages led to an intermediate version where a global memory of object structures coexisted with a working memory of logical assertions: only the latter was accessible to the inference engine. The OOPS-engine strives towards full integration of object-oriented and rule-based representation. On the one hand, rules should refer to and work directly on the hierarchy of classes and objects present in the global memory. On the other hand, rules themselves, their antecedents and their conclusions, as well as the other task-implementation structures needed to operate the inference engine (ex. the current conflict set) are all represented as objects.

• ***a powerful rule representation language***

To give an idea of the expressive power of the representation language used, the basic rule condition template is the following:

```
(<object> is <class> with
        <slot1> <operator1> <value1> &
        .. &
        <slotn> <operatorn> <valuen>)
```

where <operator> can be =, <>, <, >, <=, or >=.

This rule condition is the object-oriented equivalent of a set of two-place logical assertions where <slot> is the predicate, <object> the first argument and <value> the second argument. Note that here, the rule interpreter allows variables not only for the arguments but also for the predicate, thus approximating the power of second-order predicate logic. In addition, a rule condition may include any function call:

```
(CALL <function> <argument>* <operator> <result>)
```

Universally and existentially quantified rule conditions are also allowed:

```
(FORALL (<variable>*) (<var-domain-condition>*)
                      <subcondition>*)
(FORSOME (<variable>*) (<var-domain-condition>*)
                      <subcondition>*)
```

Finally, any of the above rule conditions (or any conjunction of these) may be negated.

The most important rule actions consist in creating or destroying objects, or modifying slot values of objects. In the general case, these actions are applied to objects of the application domain; they are then base-level actions. But since rules themselves are represented as full-fledged objects, these same actions become meta-actions when applied to rules; that is, the same set of commands allows us to change rule conditions and actions, rule priorities, etc. Furthermore, metalevel-specific actions, like activating a particular rule or rule set, allow the user to modify the inference engine's control strategy. Finally, to enhance the system's flexibility, any user-defined function can be called from a rule's right-hand side.

- ***possibility of concurrent multi-expert reasoning***

Up to this point, we have referred to 'the' inference engine as though only one were possible in a given system. Actually this system has been designed with a view to implement multi-expert systems. Any number of inference engines can be created, each associated with its own knowledge base (KB) (tutors, experts, coach). We originally envisaged a non hierarchical set of agents. Coordination would then have required a central control process such as a blackboard. However, the society of agents in ETOILE is now highly structured: the coach controls the tutors and the tutors control the experts. We hence abandoned our preliminary attempts to apply a blackboard approach.

- ***full computational support of metareasoning***

Since rules are represented in exactly the same formalism and within the same global memory as domain objects, rules can reason about rules and can examine their own (and other rules') components, evaluation contexts, history of past successes and failures, etc. Rules can also modify (even destroy) themselves and other rules as well as create new rules. Full metareasoning power is not available in classical production system languages: an OPS5 rule, for example, cannot directly examine its own instantiations because these are represented differently and separately from domain knowledge. To implement metalevel reasoning in such systems, programmers usually have to code special-purpose functions to retrieve metaknowledge inaccessible to rules.However, these facilities are currently not exploited by ETOILE.

III.2. Main principles of the inference engine

Like all production system interpreters, the OOPS inference engine follows a deliberation-action cycle.

III.2.1. Deliberation cycle

During the deliberation cycle, the interpreter evaluates and selects the rule to be fired. Two contending philosophies underlay rule evaluation procedures as implemented in existing inference engines. The first is what we might call ***maximal fact propagation***. Whenever a modification is made in the knowledge base, this change is propagated in all existing rules so that at each instant, the interpreter knows exactly which rules will fire, and with which instantiations. This approach is illustrated by the RETE network which is at the heart of many production system interpreters such as OPS5.

However, several criticisms of the Rete algorithm have been aired since. First, critics of the maximal propagation approach contend that it is at its best when working memory does not change very much. TREAT claims to remedy precisely this drawback of the OPS5 interpreter. Second, the OPS5 interpreter does not allow partitioning of the rulebase into rule sets, precisely because the RETE network is a monolithic network which has to be updated integrally each time a working memory element is added, removed or modified. Third, both the representation of rules and the operation of the RETE network are such as to make metareasoning nearly impossible. In OPS5, for example, there is no way to write a metarule that examines, for instance, the conditions, actions, current instantiations or past history of a given rule in order to take a strategic decision on rulebase management.

The solution adopted for the OOPS engine aims at overcoming these three shortcomings. For a highly volatile working memory, a partial propagation approach coupled with complete rule evaluation after conflict resolution may be the optimal solution. Furthermore, changes in working memory entail reconsideration, not of all the rules in the current rulebase, but only of those in the active rule set(s). This saves a lot of computational time in the case of very large rule bases. Finally, since rules are represented as objects in OOPS, rules can examine and modify other rules in the same way that they examine and modify any base-level object.

(1) Conflict set maintenance

When a rulebase is run, the conflict set (CS) is initialized to all the rules of the currently active rule set. Conflict set updates take place whenever changes are made to working memory, whether during rule firing (execution of a rule's right-hand side) or in response to top-level user commands which modifies working memory. When a value is added to an object slot, all rules containing a positive condition which partially match this object-slot-value triple are added to the conflict set. When a value is deleted from an object slot, all rules containing a negative condition which partially matches this object-slot-value triple are added to the conflict set.

The update technique used above, based on partial matches, is the heart of the minimal propagation approach. In the maximal propagation approach, each working memory modification leads to a complete revision of the status of all rule conditions, thus the updated conflict set contains only firable rules, i.e., rules that will surely fire. In OOPS' minimal propagation approach, the conflict set contains candidate rules, only a subset of which will surely fire. The underlying principle in the latter case is that it is safer to leave in the CS rules that will not actually fire (the price to pay would be a loss in efficiency) rather than to eliminate from the CS rules that could have fired (in which case the - unjustifiable - price to pay would be incorrectness).

(2) Rule choice

As we have seen above, the conflict set contains rules that will certainly fire as well as a few rules that may not fire. If the conflict set contains only one candidate rule, then that rule is evaluated. Otherwise, conflict resolution is based on two criteria:

- Choose the rule with the highest priority. The user can attribute priorities to rules using the keyword priority followed by an integer. The value 0 represents the highest priority. By default, rules are assigned priority 99.
- Among rules of equal priority, choose the most constrained rule, i.e., the rule with the greatest number of rule condition tests.
- If more than one rule remains in the conflict set after the above elimination process, select the first rule in the set.

(3) Rule evaluation

Rule evaluation consists in verifying the satisfaction of all conditions of the rule's left hand side by matching each one against working memory. At any moment during rule evaluation, the interpreter maintains two stacks of rule conditions—matched conditions and pending conditions—as well as the list of current variable bindings, which will constrain the possible instantiations of pending conditions.

The order in which conditions are matched is determined dynamically, on the basis of the current state of working memory. The principle of condition choice is the following: give priority to the condition that is least likely to match. The rationale of this principle is that if the rule will not fire, we might as well find out as soon as possible. That would save us computer time spent on evaluating conditions that match before having to give up on encountering one that doesn't. Below are criteria followed for dynamic condition choice:

- If there is a single remaining condition, evaluate it; else:
- Take any ground—i.e., fully instantiated—condition if one exists.
- Identify conditions that relate to the same object as that instantiated in a previously evaluated condition.

  If no such condition exists, take any condition which is completely instantiated except for the variable represented the object.

  If only one such condition is pending, then evaluate it, else select one using the following criteria:
- Select the most constrained condition, i.e., that which has the fewest unbound variables. If several conditions have the minimal number of unbound variables:
- Select the condition that is least likely to be instantiated, i.e., the condition which has the shortest list of candidate instantiations (determined on the basis of the current facts).
- If none of the above criteria yields a single candidate condition, take the first pending condition.

III.2.2. Action cycle

All actions in the current rule's right hand side are simply executed in sequence. The OOPS-engine follows the principle of rule exhaustion—i.e., once a rule has been selected, it is fired for all possible instantiations. Concretely, this means that when a rule is selected for evaluation, each of its conditions is attributed a list of possible instantiations on the basis of existing facts in the object hierarchy. The rule is fired for all possible combinations of these different instantiations. Once a rule has been fired for a given instantiation, the interpreter backtracks over the successfully evaluated rule conditions: each condition in the matched-conditions stack is unstacked and reevaluated for remaining elements in its list of candidate instantiations.

The task of finding all possible instantiations of a condition set at a given time becomes more complicated in the case of ***self-referencing rules***. A self-referencing rule is one which tests a slot of object in the left-hand side, then modifies or deletes the slot or the object in its right-hand side. When the rule is fired for a given instantiation, modifications made to a subset of this instantiation can make all pending instantiations invalid. Thus the need to unstack all evaluated conditions down to that condition whose terms have been modified during rule firing. All this is ensured by the inference engine in a manner transparent to the user. At rule compilation time, the parser identifies conditions concerning slots and objects that are affected (added, deleted or modified) by one or several rule actions. It does this by examining the correspondences between slots and objects tested by object-slot-value conditions in the LHS and the arguments of object-centered actions in the RHS. Thus the interpreter knows, at rule evaluation time, which conditions have had their instantiation candidates invalidated during firing. In order that this process can work with no major problems, the golden rule the user should follow is: a function called from the RHS of a rule should NEVER execute a hidden object-slot-value modification that will affect an object slot that is being tested in the condition part of that same rule. All such modifications should be made explicit in the RHS using typical object-centered actions such ADD-VALUE, DELETE-VALUE, SET-VALUE, MAKE-OBJECT and KILL-OBJECT.

III.3. Adapting OOPS to ETOILE

There are many differences between a genuine expert system and the role of an expert in the architecture of ETOILE. These differences led us to adapt the inference engine in various ways. Most of these differences have been defined outside the core inference engine (in a function called do-etoile-kludge).

III.3.1. Covering all solutions: optimal rules, suboptimal rules and repair rules

A genuine expert system has to produce ONE solution, ideally the optimal solution. In ETOILE, the expert must cover several solutions since he must maintain the interaction with the learner even if this learner follows a sub-optimal approach. The expert must accept a learner step which is not optimal, but still correct. Hence the expert needs both optimal rules for proposing the best actions and suboptimal rules for accepting suboptimal learner actions. This trade-off between accuracy and flexibility is obtained by attaching a priority integer to rules (see section III.2.1. on page 29). Hence, optimal rules will be considered before suboptimal ones.

The actions of the learner may be worse than suboptimal, they may be wrong. In principle, an autonomous expert system never generates an incorrect problem state since all its rules define only legal moves. However, in ETOILE, the expert might face an incorrect situation created by the learner. He therefore has a set of rules called 'repair' rules (see section 4.4. "The learner-expert-tutor triangle" on page 11). The repair rules are not triggered in isolation from the other rules. The 'repair rule' class is a sub-class of the 'rule' class. Its syntax is similar except that repair rules have an additional slot that includes a pointer towards a hypertext node that is relevant for the type of mistake that made a repair necessary.

III.3.2. Diagnosis: rule preview

A genuine expert system does not perform any learner modelling. This function implies changes in the normal sequence of operations of a production system. Here are some examples of modifications that are specific to the diagnosis mode:

- When the expert activates his rules to evaluate the learner's actions, he must find the activatable rules without activating them really. Hence, the inference engine must support a 'preview' mode that does not activate the rule conclusions.
- The learner's actions change the problem from state N to state N+1. If the expert rule preview is performed after the learner's action, he obtains the rules activatable for the state N+1. It would of course be nonsense to compare the learner's action at state N with the expert's action at state N+1. Hence, the expert previews its rulebase before the learner does anything, and stores the activatable rules for later comparison with the learner action.
- Let us assume that on one hand the learner creates a list of 20 words with no particular properties. On the other hand, one of the expert's activatable rules leads to a list of 22 words without particular properties. The learner's and expert's actions are not identical, but from a pedagogical viewpoint, they should be considered as equivalent. Hence, for each feature of an object, the author must specify a predicate (equivalent-? feature value-1 value-2) that determines when the difference between two

values of this feature is significant.

III.3.3. The integration of procedures

In ETOILE, the expert does not work alone but in interaction with a learner. He does not work in an abstract world defined by facts but he has to accomplish actions affecting the rest of the system. In other words, if one wants to integrate an inference engine into an interactive system, we must tolerate function calls in the rule actions or conditions.

Rule actions modify the objects base via OOPS-specific terminology (e.g., ADD-VALUE, DELETE-VALUE, KILL-OBJECT) which, in addition to changing some slot value, propagate these changes to determine how to update the conflict set. If a rule action includes a procedure that changes some objects with the usual Lisp accessors, these changes will remain invisible to the inference engine as it updates the conflict set. As a result, the conflict set will no longer correctly reflect the actual state of object memory. The technical solution consists in using a table where the customized functions used by the expert are explicitly declared to produce side effects on specific slots of specific classes.

In the condition part, two types of function calls are performed. The first case concerns predicates that do not exist in OOPS or summarize in one expression a condition that would need many expressions in the OOPS syntax. The second case returns particular predicates representing learner responses to questions asked by a tutor or expert (e.g. “Do you agree that I undo your previous move?”). In both cases, the function calls do not change the objects, they only affect the dynamic evaluation of conditions.

III.3.4. Control among agents

Control flows from one agent to another. Regarding the tutors and the coach, the deliberation-action cycle is repeated until it activates a rule that explicitly activates another agent. The expert cycle is determined by the two factors previously described in section 4.2. “How to implement the interaction between the learner and a computational agent?” (page 9): the local and the global interaction modes. If the local interaction mode is ‘observe’, the expert performs a rule preview, waits for the learner’s action, compares it to the actions he would have done, determines the diagnosis value (positive, negative or unknown) and returns control to the active tutor. If the local interaction mode is ‘propose’, the expert will perform a certain number of consecutive steps. The number of steps is determined by the value of the global interaction mode. In other words, the inference engine includes a simple step counter and stops when the required number of steps has been performed. A ‘step’ does not correspond to a single iteration of the deliberation-action cycle, it involves a sequence of cycles which terminates once a fired rule executes a microworld command. The expert may first fire a sequence of rules for updating private objects (e.g. task goals), and only later on activate one rule that affects the microworld. When N steps are performed successively, the expert conducts a brief dialogue with the learner between two steps. He instantiates and presents the comments associated with the fired rule, then asks the learner whether he allows the expert to carry on. If the learner agrees, the expert performs the next step. If the learner asks to stop or to continue by himself, the expert returns control to the tutor.

Appendix IV:
The Design Metaphors

Designing an intelligent learning environment (ILE) involves implementing some theory of learning and teaching. However, most available theories do not have the level of operationality required for implementation work.The design of an ILE is a creative process. Our challenge is to develop a framework that builds a bridge between theories and implementation. This framework translates psychological knowledge into terminology more relevant to computer scientists. It specifies the cognitive architecture of MEMOLAB and it is based on two concepts: the pyramid metaphor and the language shift mechanism.

IV.1. The metaphors

The ***pyramid*** is a visual metaphor for the core structure of a learning environment. The pyramid represents the concepts and skills to be acquired by the learner, ranked bottom-up according to their level of “hierarchical integration” (see below). Learning consists in moving up in the pyramid. Each level of the pyramid is defined by two languages: the ***command language*** and the ***description language***. The command language vocabulary is the set of elementary actions that the learner is allowed to do at some stage of interaction. The command language syntax defines how the learner composes sequences of elementary actions. The description language is the set of symbols (strings, graphics,...) used by the computer to show the learner some description of his behaviour. This description reifies some abstract features of the learner’s behaviour in order to make them explicitly available for metacognitive activities (Collins and Brown, 1988).

The command and description languages are different at each level of the pyramid. The hierarchical nature of a pyramid implies that each level integrates its lower neighbour. This integration is encompassed in the relationship between the languages used at successive levels: if a description language at level L is used as a new command language at level L+1, then the learner is compelled to use explicitly the concepts that have been reified at level L. This is what we called the ***language shift*** mechanism (Dillenbourg, 1992): when he receives a new command language, the learner must explicitly use the concepts that were implicit in his behaviour. The meaning of the new commands has been induced at the previous level by associating the learner’s behaviour with some representation. This representation is now the new command. The ILE structure can then be described as a sequence of [*action language, representation language*] pairs, a sequence in which the relationship between two successive pairs is described by the language shift mechanism.

Let us consider a simple example on solving equations. At some level of the pyramid, one can show the learner - with some graphics - that a good heuristic is to regroup the X’s on the same side of the equation. At the next higher level, we can offer a button “regroup X” in order to compel him to explicitly use this heuristic in her solution. A more complete example will be described in section IV.3. "Applying design metaphors to MEMOLAB" on page 34.

IV.2. Theoretical background

The role of these metaphors is to translate psychological concepts into design principles. This metaphorical link between theories and design is bidirectional and complex. By bidirectional, we mean that, in one direction, it helps the designer to ground design into a psychological theory, and also that, in the reverse direction, the empirical data collected from using the system constitute a feedback for the theory. This link is complex because it may be used to refer to several theoretical approaches. Most psychological theories address actually only a specific facet of learning while an ILE designer must consider learning in its globality and complexity. For instance, an ILE must account for the importance of discovery, for the role of practice and for the effect of coaching, because all of them occur at some stage of learning in the real world. Therefore, an intermediate framework should integrate multiple theories, each relevant for some aspect of reality. Hare are some examples of links between theories and our design metaphors.

(1) Piaget (1971) uses the term ***reflected abstraction*** for the process by which properties that are implicit at some level of knowledge can be abstracted and explicitly reached at the higher level. The language shift principle translates this idea: the designer first introduces abstract features in the de-

scription language, and at then, at the next level, coerces the learner to handle explicitly this features by using them in the command language.

(2) The pyramid and language shift concepts can be articulated with Vygotsky's idea of ***apprenticeship***. When the learner performs at some level L, the description language describes his activities at level L+1. This level L+1 corresponds to the "zone of proximal development" concept (Vygotstky, 1978). Wertsch (1985) proposed a linguistic analysis of the internalization process that relates it to the language shift. He observed (in mother-child interactions) that moving from the inter-individual to the intra-individual plane was preceded by a language shift inside the inter-individual level: mothers replace a descriptive language by a strategy-oriented language (i.e. a language that refers to objects according to their role in the problem solving strategy).

(3) During this research we concentrated on one theory: the ***neo-Piagetian theory of Robbie Case (1985)***. We focused on this theory because of it's rather operational form, and because, s***ince this theory defines developmental stages, it concerns the macro-structure of Memolab***. By macro-structure, we mean the relationship between successive microworlds. Some other theories would be more relevant for the micro-structure, i.e. the interaction among agents inside a microworld.

The key idea in Case's theory is what he calls an "***executive control structure***". In the eighties, the dominant view in cognitive science was to consider problem solving as the execution of schemata. Case discriminates "figurative schemata" which represent states (from the problem situation until the solution situation) and "operative schemata" which represent transformations. Coherent with the Piagetian tradition, he defines several stages of development: (1) perception of objects and motor activities; (2) relations between motor activities; (3) manipulation of dimensions (quantifiable variables) and (4) second order dimensions (ratios). Within each stage, Case distinguishes sub-stages where the subjects apply increasingly complex schemata. This complexity is defined by the number of "basic units of thought". Each sub-stage is characterized by the subordination of a new basic unit to the executive control structure: the first sub-stage has two basic units, the second has three and the third has four. The most interesting aspect of his contribution concerns the ***transition between stages***. The complexity of schemata reached at the last sub-stage of stage N corresponds to a basic unit at stage N+1. The four-unit control structure of stage N is turned into a one-unit control structure at stage N+1. This process is called "***hierarchical integration***".

According to Case, the ability to apply increasingly complex schemata is mainly due to the evolution of the subject's working memory (that Case calls "Short Term Storage Space"). From a developmental perspective, working memory increases with age as a result of the maturation of the nervous system. In addition, intensive practice of schemata leads to their automation or compilation, freeing up memory resources for additional basic units. The original contribution of Case is the understanding of why learning is not a simple linear process. It includes ***quantitative changes*** within a stage resulting from practice effects, to ***qualitative changes*** between stages resulting from knowledge restructuring.

There is an obvious mapping between the structure defined by Case and our intermediate framework. The control structures at each level of the pyramid integrate the control structures located at the lower level. The sequence of microworlds within the pyramid is structured along Case's view of development: quantitative variations define possible improvement within some level (or microworld or stage) while the qualitative variations define the transition between two levels. The concept of stage transition is translated into the language shift mechanism. This transition is necessary when the learner tries to solve problems that have too high memory load constraints. After the language shift, the learner has at his disposal new control structures that enable him to solve the problems with a reduced cognitive load. This ILE-oriented re-expression of Case allowed us to ground the design of MEMOLAB in this theory.

IV.3. Applying design metaphors to MEMOLAB

MEMOLAB is structured according to the principles described above. The MEMOLAB pyramid has three levels. Creating an experiment means assembling experimental 'building blocks' on the lab workbench. These 'blocks' or units are 'events' at the first level, 'sequences' at the second level and 'plans' at the third level. An event (level 1) is the association of some subjects with some task and a measurement device. A task is defined by an activity, a material and some parameters. A sequence (level 2) is the set of activities performed N times by a same group of subjects, with different materials equivalent to the material specified by the learner. A plan (level 3) is a table of factors and modalities that will instantiate a sequence in which some parameters have been replaced by a variable.

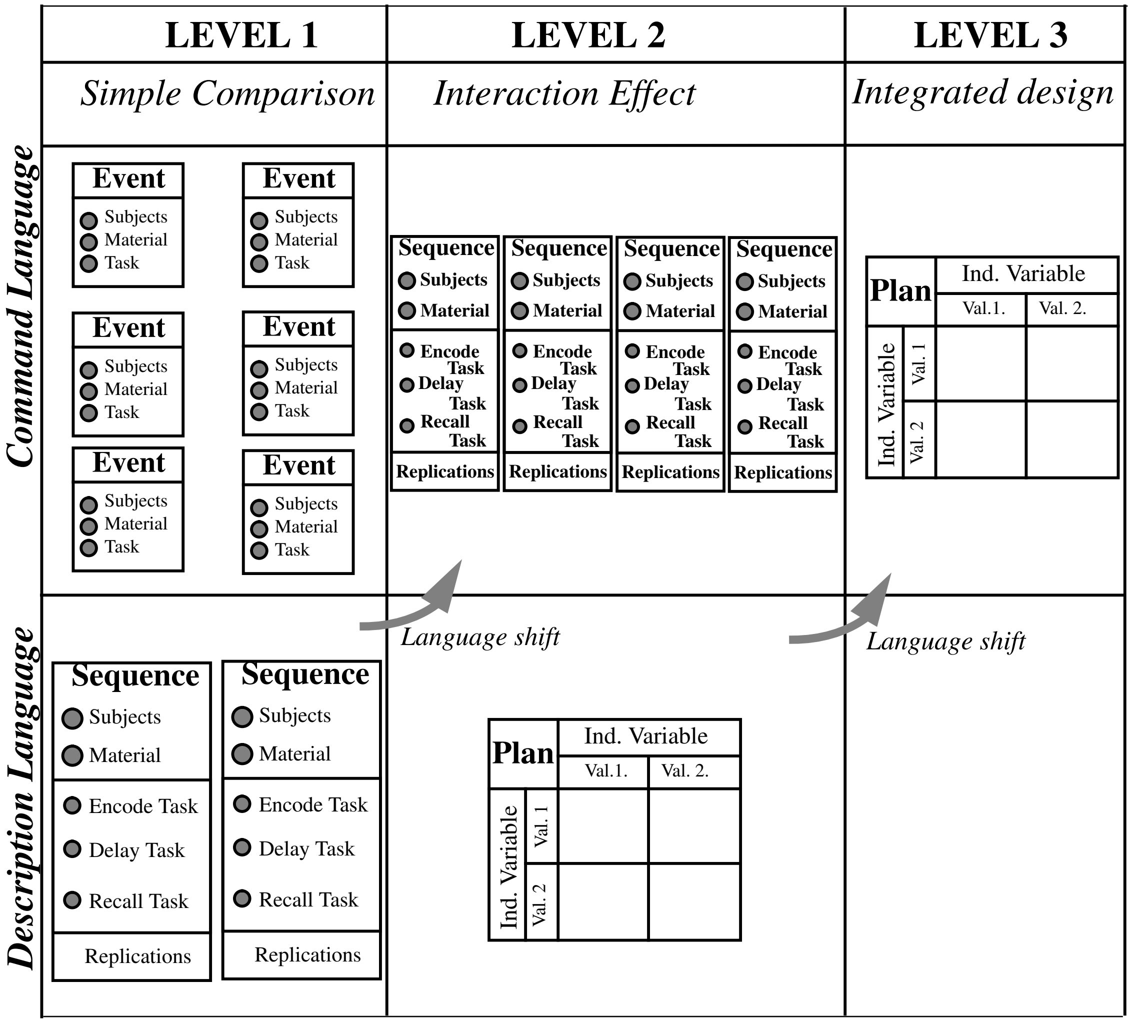

Figure 14: The Cognitive Architecture of MEMOLAB

An overview of the learner's activities at each level of the pyramid is presented in Figure 14. For each level, we represent two screens, one displaying the command language and the other presenting the description language. At level one, the novice sequences events on the lab workbench. At the end of level 1, the learner receives challenges that induce the need for comparisons. The concept of sequence is reified at level 1 and used at level 2 as the way of designing new experiments. At the end of the second level, the necessity of taking into account the interaction of effects is induced by new challenges. The concept of plan is presented at level two and used for designing experiments at level 3.

The shift from one level to another, i.e. shifting from one language to another corresponds to some *qualitative* jump in learning. Within each level, we vary in a more continuous way the difficulty of the challenges proposed by the tutor. The last challenges of level 1 already have the difficulty of level 2. This shows the learner the necessity to have more powerful control structures to solve the proposed challenge. (As in Case theory sub-level i.4 is equivalent to sub-level i+1.0). The difference between levels depends not only on the number of parameters in the experiment but also on more subtle aspects of the task. For instance, at level 2, the described experimental sequence must be considered as the abstraction of something that must be replicated N times. The number of replications varies according to the duration of each experiment in order to compromise between the subject's tiredness and the need for multiple measurement. Another tricky part of the expertise, the selection of interesting parameter values, is brought into the game at level 3.

9. Bibliography

ANDERSON, J.R (1984) Cognitive Psychology and Intelligent Tutoring. P*roceedings of the Cognitive Science Society Conference*, Boulder, Colorado, pp. 37-43.

BAKER, M. (1992) *The collaborative construction of explanations.* Paper presented to "Deuxièmes Journées Explication du PRC-GDR-IA du CNRS", Sophia-Antipolis, June 17-19, 1992.

BULL, S., PAIN, H. & BRNA, P. (1993) Collaboration and Reflection in the Construction of a Student Model for Intelligent Computer Assisted Language Learning. *Proceedings of the Seventh International PEG Conferenc*e, Edinburgh, July 1993.

CASE, R. (1985) *Intellectual Development form Birth to Adulthood.* New York: Academic Press.

CAWSEY A. (1993) Planning Interactive Explanations. *International Journal of Man-Machine Studies*, 38, 169-199.

CLARK, H.H. & BRENNAN S.E. (1991) Grounding in Communication. In L. Resnick, J. Levine and S. Teasley. *Perspectives on Socially Shared Cognition* (pp. 127-149). Hyattsville, MD: American Psychological Association.

COLLINS, A and BROWN, J. S. (1988) The Computer as a Tool for Learning through Reflection. in H. Mandl and A. Lesgold (Eds), *Learning Issues for Intelligent Tutoring Systems.* (pp. 1-18). New York*: Springer* Verlag.

DILLENBOURG, P. (1992) The Language Shift: A mechanism for Triggering Metacognitive Activities. In M.Jones and P. Winne (Eds.), *Adaptative Learning Environments* (pp. 287-316). Berlin: Springer Verlag.

DILLENBOURG, P. (To appear) Distributing Cognition over Humans and Machines. in S. Vosniadou, E. De Corte, R. Glaser and H. Mandl (Eds) *International Perspectives on the Psychological Foundations of Technology-Based Learning Environments*. New York: Springer-Verlag.

DILLENBOURG, P. & MENDELSOHN, P. (1992) The Genetic Structure of the Interaction Space. In E.Costa (Ed.), *New Directions for Intelligent Tutoring Systems* (pp 15-27). Berlin: Springer-Verlag.

DILLENBOURG, P. & SELF, J.A. (1992a) A framework for learner modelling. *Interactive Learning Environments*, *2*(2), 111-137.

DILLENBOURG, P. & SELF, J.A. (1992b) A computational approach to socially distributed cognition. *European Journal of Psychology of Education*, 3 (4), pp. 353-372.

ELSOM-COOK M.T. (1991) Dialogue and teaching styles. In P. Goodyear (Ed). *Teaching knowledge and intelligent tutoring* (pp. 61-84). Norwood (NJ): Ablex.

FREDERIKSEN, J. & WHITE, B. (1988) Intelligent Learning Systems for Science Education. *Proceedings of the Internatioal Conference on Intelligent Tutoring Systems.* ITS-88. Montreal, Canada. June. pp. 250-257.

JONES M. & WIPOND K, (1991) Intelligent environments for curriculum and course development. In P. Goodyear (Ed). *Teaching knowledge and intelligent tutoring* (pp. 379-395). Norwood (NJ): Ablex.

MERRILL M.D & LI Z. (1989) An instructional Design Expert System. *Journal of Computer-Based Instruction*, Vol. 16, 3, pp. 95-101.

MOYSE, R. & ELSOM-COOK, M. (1992) *Knowledge Negotiation*. London: Academic Press

NEWMAN, D. (1989) Is a student model necessary ? Apprenticeship as a model for ITS. *Proceedings of the 4th AI & Education Conference* (pp 177-184), Amsterdam: IOS.

PIAGET, J. (1971) *Biology and Knowledge*. Chicago: The University of Chicago Press.

RIESBECK, C. K. and SCHANK, R. C. (1989) *Inside Case-Based Reasoning*. Hillsdale: Lawrence Erlbaum Publishers

ROGOFF, B. (1991) Social interaction as apprenticeship in thinking: guided participation in spatial planning. In L. Resnick, J. Levine and S. Teasley. *Perspectives on Socially Shared Cognition* (pp. 349-364). Hyattsville, MD: American Psychological Association.

ROSCHELLE, J. (1990) Designing for Conversations. Paper presented at the AAAI Symposium on Knowledge-Based Environments for Learning and Teaching, March. Stanford,CA.

ROSCHELLE, J. (1988) Sufficiency and utility of physics problem solving processes. *Proceedings of the Internatioal Conference on Intelligent Tutoring Systems (ITS-88).* Montreal, Canada. June. pp. 132-139

RESNICK L., LEVINE J., and TEASLEY S. (1991) *Perspectives on Socially Shared Cognition.* Hyattsville, MD: American Psychological Association.

RUSSELL D.M., MORAN T.P., JORDAN D.S. (1988) The Instructional-Design Environment. in J. Psotka, L.D. Massey and S.A. Mutter (Eds) *Intelligent Tutoring Systems. Lessons Learned.* (pp. 203-228). Hillsdale, NJ. Lawrence Erlbaum

SCHRAGE M. (1990) *Shared Minds. The new technologies of collaboration.* New York: Random House

SOHAM, Y. (1993). Agent-oriented programming. *Artificial Intelligenc*e, 60, pp. 51-92.

TERVEEN, L.G.. WROBLEWSKI, D.A. & TIGHE S.N. (1991) Intelligent Assistance through Collaborative Manipulation. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-91).* pp. 9-14. Sidney, Australia.

VYGOTSKY, L.S. (1978) *The Development of Higher Psychological Processes* (edited by M.Cole, V. John-Steiner, S.Cribner and E. Souberman). Harvard University Press. Cambridge. Mass.

WERTSCH, J. (1985) Adult-Child Interaction as a Source of Self-Regulation in Children. In R. Yussen (Ed). *The growth of Reflection in Children* (pp. 69-97). New York: Academic Press.

WINNE P. & KRAMER L.L. (1988) Representing and inferencing with knowledge about teaching: DOCENT- an artificially intelligent planning system for teachers, *Proceedings of Intelligent Tutoring Systems (ITS-88)*, (pp.7-15), Montreal, Canada.