

Article

A Hybrid Expert System as an Embedded Module in Tutoring Systems

I. D. Zaharakis^{1,2}

email: john@math.upatras.gr

A. D. Kameas²

email: kameas@math.upatras.gr

P. E. Pintelas^{1,2}

email: pintelas@math.upatras.gr

¹ Division of Computational Mathematics & Informatics, Department of Mathematics, University of Patras, Hellas

² Educational Software Development Laboratory, Department of Mathematics, University of Patras, Hellas

Abstract

Every Intelligent Tutoring System (ITS), in order to justify the term 'intelligent', has to contain the specific knowledge of the taught subject as well as to embed several modules with those mechanisms that are able to handle the knowledge in an intelligent way (e.g., using rules of thumb, tricks etc.) and, in general, behave in a way that would be considered intelligent if observed by a human. Such modules are apparently necessary in ITS generators too. In this paper, the Hybrid Methodology Tutor (HMeT) is presented. HMeT is a hybrid expert system that is embedded in an ITS generator and supports the description of the structure and dynamics of the procedural knowledge, during authoring; it is also responsible for its presentation during tutoring. HMeT was designed and implemented to replace an earlier version (MeT - Methodology Tutor), in order to provide for higher inferential capabilities, better

user interface and more user friendliness in authoring as well as in tutoring. HMeT functions like an expert system shell for knowledge base generation, provides graphical user interfaces and prototyping facilities for fast application development and uses frames for representing the subject that will be taught and rules for inference.

Introduction

During the past two decades, Intelligent Tutoring Systems (ITS) have been designed and developed in an attempt to overcome the shortcomings and inadequacies of traditional CAI systems. ITSs include knowledge-based modules and require knowledge-engineering skills from authors. Consequently, one of the clearest ways to introduce experienced teachers in building ITSs is to build easy-to-use ITS generators (Woolf, 1987).

The major design issue that developers of ITS generators face is the provision of mechanisms capable of encoding the requirements of the different components of the tutoring process. Mechanisms of this kind have to support the representation of what information is to be taught (the domain) and of how the training process can be carried out as efficiently as possible (the instructional strategy) (Mispelkamp, 1992). In order to design the internal structures of the knowledge domain, designers have to provide representation mechanisms for two kinds of knowledge (Anderson et al, 1990; Woolf, 1987): declarative and procedural. *Declarative knowledge* contains the information that the trainees have to acquire. *Procedural knowledge* consists of rules for the application of this information to solve problems as well as guidelines on how to apply these rules.

Although most knowledge representation schemes tend to favour one kind over the other, elements of both must be included in every knowledge-based system that is actually developed. The form in which this knowledge is stored determines the ways it can be used, although authoring systems tend to hide internal representation from authors. No general form suitable for representation of all types of knowledge exists (Rickel, 1989), and designers of ITS generators have to choose among alternatives, that range from making authors use AI programming languages and techniques, to providing an informative interface with limited generality of application.

This paper presents the design and the functionality of the Hybrid Methodology Tutor (HMeT), a hybrid expert system that is embedded in GENITOR.

GENITOR (Kameas and Pintelas, 1997) is an ITS generator, that provides authors with tools that may be used to completely represent the knowledge domain of a training application, while ready-to-use solutions are provided for most of the authoring activities that relate to the specification of instructional strategy. It produces stand-alone intelligent training applications in subjects that are not necessarily related with each other which attempt to transfer to the trainees two kinds of skills: procedural knowledge on how to apply a certain methodology and declarative knowledge as the corresponding theoretical background to support it. In order to overcome the complexity of the authoring process that is due to the highly dynamic nature of tutoring process, GENITOR employs two expert systems: the Domain expert Tutor (DeT) and the Methodology Tutor (MeT). The former is employed during the presentation of declarative knowledge whereas the latter supports the description of the structure and dynamics of the procedural knowledge, during authoring, and is responsible for its presentation during tutoring. From the experience gained with the early version of such a system (MeT) (Zaharakis et al, 1994) it was evident the need for higher inferential capabilities, more authoring and tutoring simplicity and a graphical user interface. The design and development of HMeT, the system that replaced MeT, were addressed towards the solution of these issues.

In the next sections, a background and related work on knowledge representation paradigms as well as HMeT requirements and design that consists of knowledge representation, authoring and runtime process will be presented. Throughout this paper, the Waterfall Model development is used as an example of a methodology that can be taught by HMeT. Furthermore, the users who will develop a training application using HMeT are called 'authors', (and are not to be confused with the 'authors' of this paper), while the users of the training applications are called 'trainees'.

Background and related work

The knowledge representation schemes that have been proposed in the past are of two general types: *rules* and *structures*. The former type includes logic, in its many different forms, including propositional logic, predicate logic, temporal logic, non-monotonic logic, fuzzy logic etc., and production rules, while the latter includes associative (semantic) networks, frames, scripts and objects.

Logic has well defined syntax and semantics and provides several types of inference that are used to automate the reasoning process. Although classical logic cannot handle incomplete or imprecise information, many systems have been developed or designed that

manipulate such information using techniques like *subjective Bayesian method*, *certainty factor model*, *Dempster-Shafer theory* or *network models* (Lucas and van der Gaag, 1991). Propositional logic offers limited expressiveness and although predicate logic is much more expressive, as a monotonic reasoning formalism, is not suitable for real-world problems. Also, as mentioned in (Lucas and van der Gaag, 1991), predicate logic is in-decidable and any automated reasoning method for it may lead to an NP-complete problem.

Production systems differ from logic, as they are generally non-monotonic and they accept uncertainty in the deductive process (Gonzalez and Dankel, 1993). Production systems involve rules that determine actions based on conditions and offer a uniform model, with each rule having the ability to fire independently. Production rules have been widely adopted for domain knowledge representation, mainly because they are simple and easy for humans to understand, since they come close to the human way of reasoning. They support a declarative style of programming, seem well suited for representing heuristic knowledge and allow the explanation of the solution derived or the solving strategy followed by the system. Their uniformity, however, masks underlying cognitive processes and limits application of rules to relatively simple and strictly-structured domains, where the reasoning processes are established and clear. Consequently, they appear to be less effective in expressing more subtle forms of knowledge which can be used to reason about the fundamental nature and causes of interesting phenomena (Graham and Jones, 1988). Another disadvantage is that whenever a rule is going to be added, deleted or modified, contradictory knowledge or infinite chains may be caused (Gonzalez and Dankel, 1993).

A richer representation of the domain can be achieved by using a highly interconnected network of knowledge objects related with semantic links that represent their properties and relationships. Semantic nets (Quillian, 1968) are very good at expressing knowledge about physical or conceptual objects, class inheritance properties, defaults and perspectives, and consequently they are commonly used to structure more general kinds of information. Procedures can then be used to traverse the network along certain links and dynamically generate knowledge. Such an organisation can simulate learning and encode meta-knowledge and common sense knowledge, that can be used to determine the degree and quality of trainees' misconceptions. Some weaknesses of this formalism that have been identified are logical and heuristic inadequacy; in addition, it sometimes may result in combinatorial explosion (Graham and Jones, 1988; Gonzalez and Dankel, 1993).

Frames are an equivalent (in terms of representation) way of grouping information. Each frame is a record of 'slots' and 'fillers' (Minsky, 1975) and can be thought of as a complex node in a network, with a special slot filled by the name of the object that the node stands for, and the other slots being filled with the values of various common attributes associated with such an object (Jackson, 1986). The knowledge in frames is structured, and is organised hierarchically, so that the attributes of the higher classes are inherited to the lower classes. In this way, frames are able to determine their own applicability in given situations. Also, the values of the slots can be dynamically stored during execution of a frame-based system. Some heuristic inadequacy may appear in the frame formalism too. Scripts (Schank and Abelson, 1977) is a variation of frames, while objects (Stefik and Bobrow, 1984) are governed by the principles of the object-oriented approach for knowledge representation without avoiding the limitations of the frame formalism.

The inadequacies of one knowledge representation paradigm may be handled effectively by another. Several expert system tools offer hybrid representation facilities that combine the advantages of different representation techniques. For example, KAS, which was derived from PROSPECTOR by removing the domain knowledge base, and AL/X, use a combined rule-based and semantic network representation technique; systems like CONPHYDE and AIRID were then developed using KAS, while AUDITOR was developed using AL/X. LOOPS, combines an object-oriented representation with a rule-based scheme; PALLADIO and its embedded system TRANSISTOR SIZING SYSTEM were developed using LOOPS and as mentioned in (Waterman, 1986), ACES is implemented in LOOPS too. KEE supports frames, objects and rules in combination, while TQMSTUNE and COMPASS were developed using KEE. The hybrid expert system CENTAUR is another system in which a combination of frames and rules is used for knowledge representation. Hybrid systems, combining frames and rules, offer a rich structural formalism with inheritance properties, control over the objects referred to and heuristic adequacy. In addition, the organisation of system's production rules into taxonomies makes it easier for the domain expert to understand and construct them and for the system designer to control when and for what purpose particular collections of rules are used by the system (Fikes and Kehler, 1985).

Design

Since applications produced with GENITOR focus on the training on some methodology, HMeT has to be used both during authoring and teaching of this methodology. A *methodology* is any procedure that

consists of distinct, partially ordered tasks, actions to carry out each task and results of each action. Such a methodology has a well-defined structure that consists of several layers of *groups of activities* (phases, sub-phases etc.) and *activities* that make up a group. Within each activity, *artifacts* are produced; these have the meaning of inputs (prerequisites) or outputs (objectives) of the activity. In order for an artifact to be produced, other artifacts must already exist. In a nutshell, the methodology is described by the authors and is stored in the procedural domain knowledge base of the application. From now on methodology and procedural domain knowledge base are used as synonymous. The trainees are then asked to order and structure the different elements that compose the methodology.

The presented hybrid expert system consists of five modules. The *Authoring Module* that is the knowledge base generator, the *Authoring Interface* where the authoring process is realised, the *Methodology module* that contains a general model of the methodology and is capable of validating the trainees' choices, the *Tutoring module* that contains a general trainee model and its role is to transmit to the Methodology module the trainees' choices and to filter the Methodology module responses depending on the trainees' performance data and the *Tutoring Interface* where the whole interaction process is realised. The Authoring Interface and the Authoring Module are involved in the authoring process, while the rest are involved in the tutoring process. A high level architectural view of HMeT is given in Figure 1. In the next, the knowledge representation paradigms handled and the system modules will be described.

Knowledge representation

For HMeT system needs, the general structure of the knowledge base has to be as abstract as possible, in order to be applicable to different in nature methodologies and in the same time be compact for authoring process simplicity. Also, a powerful mechanism for knowledge manipulation and detailed explanation has to exist. A hybrid scheme of frame objects and production rules has been used in order to represent the elements of the procedural knowledge. The system has been implemented in Prolog, so the declarative programming style has been followed and several Prolog characteristics have been inherited, like non-monotonic reasoning and backtracking. Each kind of methodology element is represented with a frame class; then, elements are instantiations of objects in the class. Element properties are represented with frame slots, while a set of predefined rules describes the way slots are handled by the system. On the frame slots, demons (Waterman, 1986) are used that take action when a slot value changes or is accessed. Frame objects per-

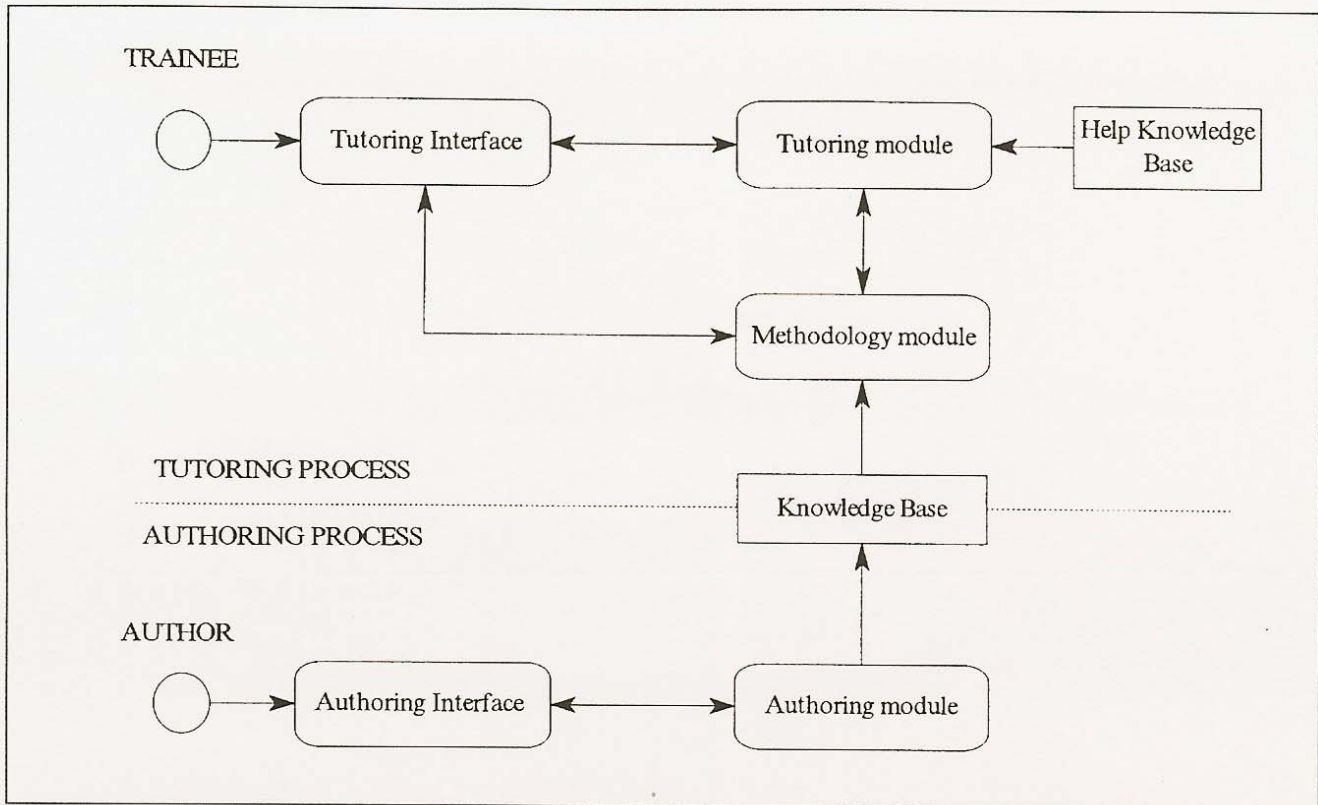


Figure 1. HMeT high level architecture

mit a straightforward visualisation of the methodology evolution using a simulation environment; simulation is considered as the best available technique to transfer procedural knowledge to the trainees (Rickel, 1989). In addition, frame objects imply authoring simplicity: authors define the elements that will be used and assign appropriate values to the slots of each frame instantiation. No programming skills or knowledge of AI is required; a good understanding of the methodology elements and a thorough design of the methodology structure are enough.

The object classes that constitute the knowledge base are *group of activities*, *activity* and *artifact*. The class *object* is the generic class. The inheritance relationship, *is_a*, between the classes is described in Figure 2. This is a hierarchical relation and is deduced by the rules of Methodology module; it can not be changed by the authors. Every class low in the hierarchy inherits the attributes of the higher classes in the same branch of the treelike taxonomy while it may have its own attributes.

Three other relations between the objects that describe the knowledge base model exist:

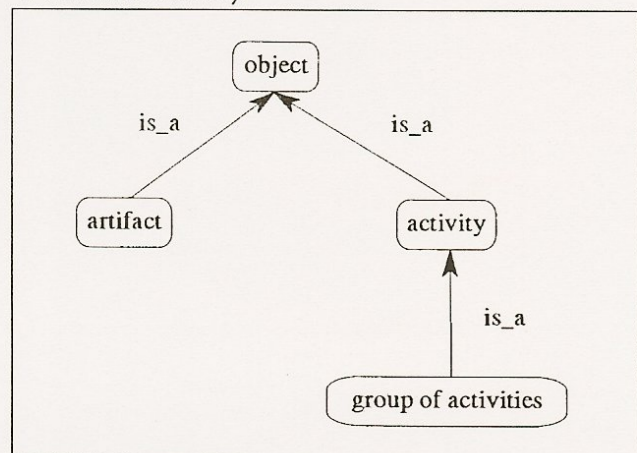
- *composed_of*, between a group of activities and one or more groups of activities, or between a group of activities and one or more activities; it indicates the nested layers inside a group of activities that are formed by several other groups of activities as subgroups or simply activities

- *produces*, between an activity and one or more artifacts; it indicates the artifacts that have been realised so that other activities may be available
- *needs*, between an artifact and one or more artifacts; it indicates the prerequisites of an artifact production, in order to ensure the continuation of the training process.

In Figure 3, the different objects manipulated by the system and the relationships between them are represented.

The frames in Figures 4, 5, 6 describe the artifact class, the activity class and the group of activities class, respectively. All kinds of frames have a *class*

Figure 2. Object classes manipulated by the system in a treelike taxonomy.



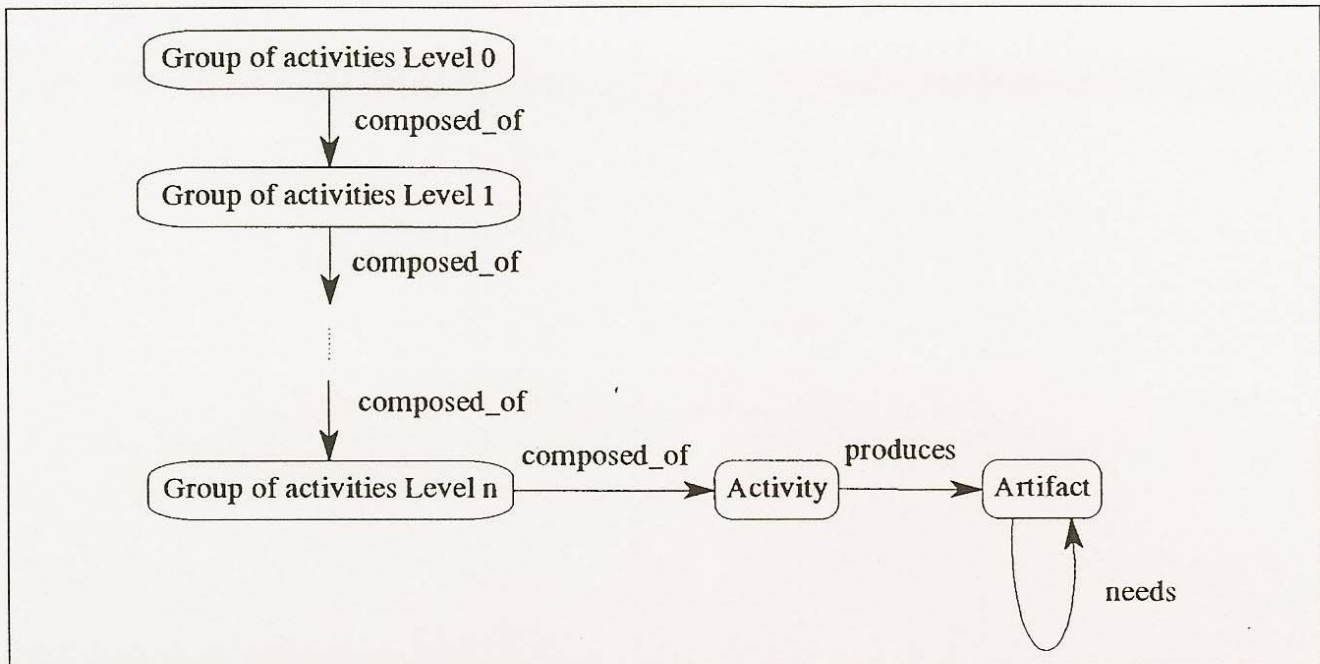


Figure 3. The relations between objects

and an *is_a* field. The value of the *class* field is the name of the class (e.g., project, phase, information etc.). The *is_a* field is an inheritance relationship and its value indicates the parent of each class. The values of these two fields are system specified and can not be changed by the authors.

All classes have three attributes in common that are inherited from the generic class object:

- *identifier* that indicates the name that the class assumes in a specific methodology; its value is set by the system in the object definition during the authoring process
- *label* that identifies the class object; its value is specified by the authors who model the methodology
- *state* that indicates the state of the class; its value is default in the beginning but is changed along the tutoring or runtime process. Permitted values for the classes activity and group of activities are 'not started', 'started', 'terminated'; for the class artifact are 'not existing' and 'existing'.

The frames of group of activities contain two additional attributes:

- *level* takes values ranging from 0 to n , $n \in \mathbb{N}$, and indicates the nesting level of each class of group of activities;
- *rank* indicates the sequence in which the group of activities is involved in the methodology; the *rank* slot is filled automatically by the Authoring Module and depends on the order the authors have written the instances of the class group of activities.

The slot *type*, contains information about the type of the artifact (e.g., text, sound, picture, video etc.) and it concerns the future development of DeT for multimedia handling.

```

class: <name of the class>
is_a: <name of the parent of the class>
— slots —
identifier: <name that the class instantiates in the methodology>
label: <instance description>
state: <instance state>
type: <instance type>
  
```

Figure 4. Artifact class

```

class: <name of the class>
is_a: <name of the parent of the class>
— slots —
identifier: <name that the class instantiates in the methodology>
label: <instance description>
state: <instance state>
  
```

Figure 5. Activity class

```

class: <name of the class>
is_a: <name of the parent of the class>
— slots —
identifier: <name that the class instantiates in the methodology>
label: <instance description>
level: <level indicator>
rank: <rank indicator>
state: <instance state>
  
```

Figure 6. Group of activities class

Demons are attached procedures to frame slots that are activated by changing or accessing slot values. They differ from rules in the sense that they are activated when a change or an access of the value of the slot they are attached to takes place; on the contrary, the applicability of rules is repeatedly tested. Because of their activation when the slot values are affected, they often are referred to as *If_Needed*, *If_Added* or *If_Removed*. Such demons (used in HMeT) like *read_frame*, that takes information from the frames slots, or *affect* that modifies the frame slot value, and their functionality, are represented in the next example; for a better understanding of the example, a small part of the knowledge base (methodology of Waterfall model) is presented too.


```
instance_of(artifact,art3)
instance_of(artifact,art2)
instance_of(group_of_activities,gact5)
instance_constant(gact5,[slot(label,['Operation &
Maintenance']),
slot(level,[1]),
slot(rank,[5])])
instance_constant(art3,[slot(label,['System Model'])])
instance_constant(art2,[slot(label,['Feasibility
Report'])])
instance_variable(art3,[slot(state,['not existing'])])
instance_variable(art2,[slot(state,['not existing'])])
instance_variable(gact5,[slot(state,['not started'])])
read_frame(Name,Attribute,Value)
e.g., read_frame('System Model',state,Value)
returns Value = ['not existing']
read_frame('Operation & Maintenance',state,Value)
returns Value = ['not started']
affect(Name,Attribute,Value).
e.g., affect('Feasibility Report',state,[existing])
changes the value of the state slot of the 'Feasibility
Report' artifact from 'not existing' to existing i.e.,
after affect firing, the knowledge base is updated to
instance_variable(art2,[slot(state,[existing])]).
```

The Authoring Subsystem

HMeT has to support the generation of the domain knowledge base in an easy and flexible way and allow its debugging and correctness-testing. The authors should be supported in describing or modifying the model of a methodology in order to teach it, and in executing this model in order to verify and validate it. The authoring subsystem contains the Authoring Interface and the Authoring Module, which are involved in the authoring process. The former is a graphical user interface through which the whole authoring process is realised in an easy-to-use way. Using it, authors describe the methodology to be taught; then the Authoring Module generates the knowledge base.

The Authoring Interface provides the necessary

utilities for knowledge base creation, testing and validation (Figures 7, 8). These operations can be executed by pressing the appropriate screen button. When the *Define Vocabulary* button is depressed, a template showing the current instantiation of the names of the classes of objects handled by the system is presented. The modification of these names is at authors' preference. These names are used by the Authoring Module during the authoring process and by the Methodology and Tutoring Modules during the runtime process. In Figure 7, the default system vocabulary is illustrated. The three windows at the right side of the screen are the space where the authors can specify the names of the classes; as the class group of activities may be composed of several layers (i.e., subgroups), the names of these classes are displayed in the window labelled 'Group of Activities'. The current instantiation of the whole structure of the methodology classes is displayed in the window labelled 'Vocabulary'.

By pressing the *Create/Modify Links* button (Figure 8), the authors can select a class that they want to instantiate. This can be done by subsequently pressing the proper button in the floating toolbox on the screen; in Table I, each button functionality is explained. Then, they drop it in the desired location on the screen and type the name that the class instantiates; the form of the pointer is changed according to the selected button from the toolbox. In order to create another instance the previous process has to be repeated. The authors need not describe the relations between the objects as it was the case in MeT; these are created automatically by the Authoring Module when an object is dragged'n'dropped on another object, as it is described in the following. However, because the artifacts have the meaning of inputs (pre-requisites) or outputs (objectives) of the activity, the authors have to declare the artifacts that initially exist, which are necessary for the commencement of the methodology; this is achieved by using the last button at the bottom of the toolbox. For the same reason, while the relations 'composed_of' and 'produces' are implied from the position of the objects in the screen, the relation 'needs' is not apparent; so, when an artifact is selected, the symbol  appears before the artifacts which are required in order for the selected artifact to be produced. The authors can check the correctness and validity of the methodology through simulation by pressing *Test the model* button.

The Authoring Module is the underlying mechanism of the Authoring Interface. It contains the general form of the manipulated classes of objects and creates the frames of the domain knowledge base that will be handled by the system. Since it contains the rules about the valid relations between classes of objects, as well, it does not allow the authors to create

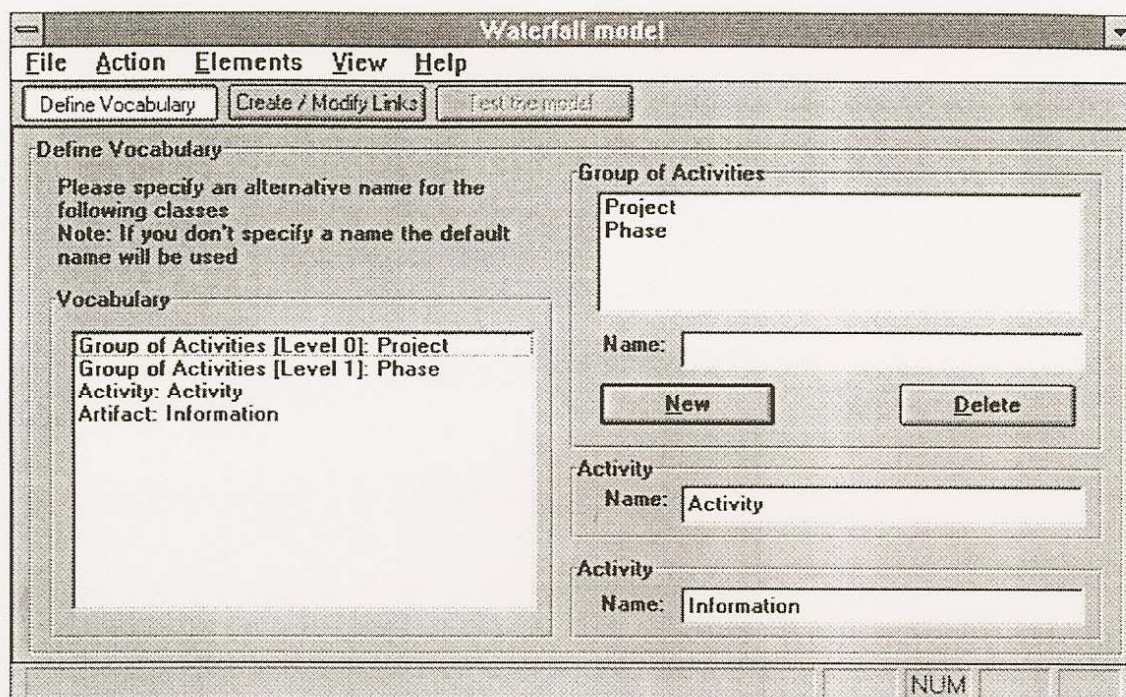


Figure 7. Authoring interface - Define Vocabulary operation

non-permissible links between objects; instead, an error message and its explanation is provided to the authors. When an object is dragged'n'dropped, the Authoring Module uses these rules to create the valid relations between objects without any further effort by the authors. All the operations in Authoring Interface, except *Test the model*, are handled by the Authoring Module.

The Runtime Subsystem and the Tutoring Process

The runtime subsystem consists of the Methodology module, the Tutoring module and the Tutoring Interface which are involved in the execution of the tutoring application. All these modules interact with each other in order to support the tutoring process. This process is realised by using the model of the methodology that the authors had described in the Authoring Interface and a trainee model that is updated during the runtime process and is displayed in the Tutoring Interface.

During the tutoring process the comments and the instructions of the several system modules have to be displayed on-line. According to (Kameas and Pintelas, 1997), tutoring systems must employ an instructional strategy which entails dialogue management that is based on the knowledge domain and makes use of a trainee model in order to execute a learning scenario; a learning scenario is considered as the context in which training takes place. Conse-

quently, the system must allow the execution of different tutoring strategies as well as the temporary interruption of the execution process and afterwards the re-entry in the environment. A trainee model that records the personal characteristics and progress in the methodology of every trainee has to be maintained. According to these, help of different verbosity has to be displayed. In addition, the trainees should be provided with help utilities, like access in a Library of units of declarative knowledge or a Glossary of terms of the specific methodology that is taught.

The Tutoring Interface (Figure 9) presents the trainee with the list of group of activities and the list of activities. The trainees have to make a choice among the elements of the lists - or properly press the button labelled 'Finish Current Group of Activities' in order to terminate an already started group of activities - that is transferred to the Methodology module in order to check its correctness. The Methodology module returns a message to the Tutoring module which has in turn to decide the answer that will be presented to the trainees as well as its verbosity. During the runtime process, the trainees' advancement is displayed in a graphical form in a separate window on the screen. The trainees are allowed to interrupt the runtime process in order to execute other system utilities like access a methodology topics library or a glossary of related terms. When an interrupt is caused the system status is saved; the re-entry in the environment is achieved in

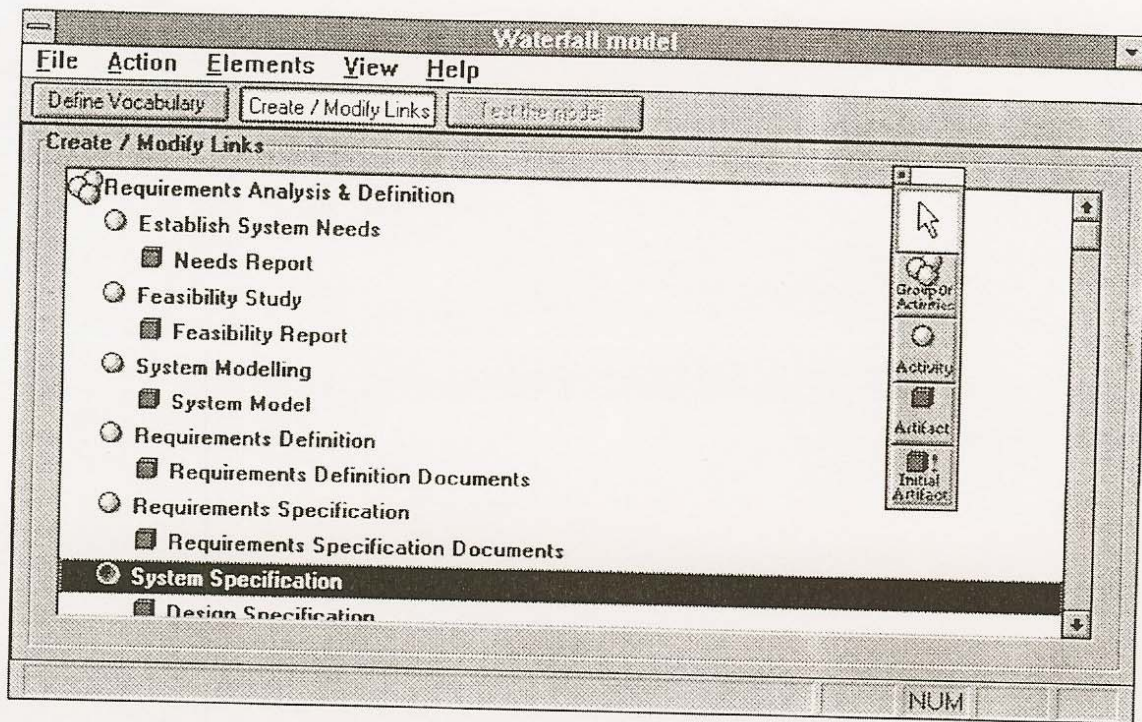







Figure 8. Authoring interface - Create/Modify Links operation

the position that the interrupt was caused. The open architecture of the presented system enables it to employ several instructional strategies that can be implemented as autonomous, external parts and can be provided as agents. Like in MeT (Zaharakis, et al,

Table I. Floating toolbox functionality

	Standard cursor form.
	The authors can create a group of activities instantiation. The relation 'composed_of' between groups is implied and created by Authoring Module.
	The authors can create an activity instantiation. The relation 'composed_of' between groups and activities is implied and created by Authoring Module.
	The authors can create an artifact instantiation. The relation 'produces' between activities and artifacts is implied and created by Authoring Module.
	The authors can create an initial artifact instantiation.

1994), three agents are used for this purpose: HMeT-Guide (G-HMeT) 'guides' trainees through a simulation of the methodology evolution by selecting itself each time the correct activity; HMeT-Coach (C-HMeT) supports a learn-by-discovery process of the methodology, and HMeT-Judge (J-HMeT) leaves control of the simulation entirely at the trainees' hands, by adopting a role of 'judge' of their selections. Every agent has its own learning scenario and the trainees have several operations at their disposal.

The Methodology module contains the rules that handle the knowledge base and produces the data for the explanation process. An object that has been realised is an object in state 'existing', for the artifact, or 'terminated', for the other objects. An object realisable is an object that, in the next action, can be in state 'started', for the group of activities and the activities or 'existing', for the artifacts. An artifact is realisable if all the artifacts that it needs have been realised. An activity is realisable if all the artifacts that the activity produces are realisable or have been realised. A group of activities is realisable if the objects, which compose it, exist and are realisable or have been realised.

The rules of the Methodology module make inferences based on the value of the frames slots and fire actions depending on these. In the following, the most important rules and their syntax in BNF form will be described; these are summarized in Table II.

Here, {charstring, stringname, integer, '!', ',', '(', ')', success, failure in the context, failure out of the context, not started, already started, already terminated,

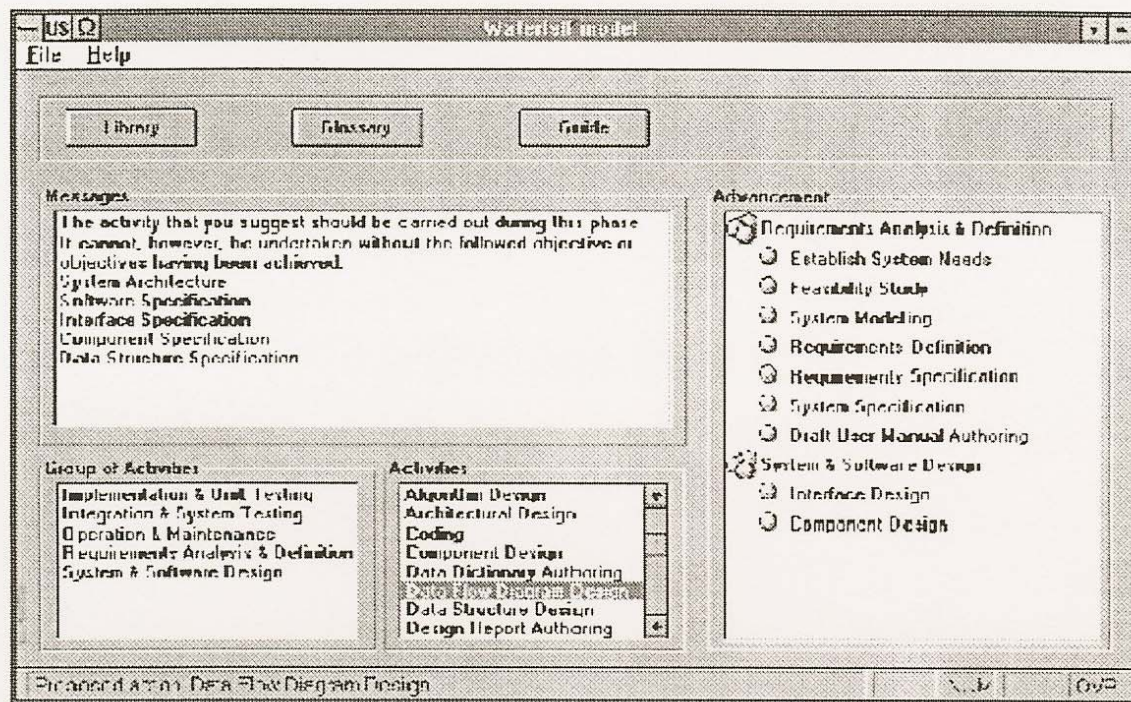


Figure 9. Tutoring interface

ARTIFACT, ACTIVITY, LEVEL) is the set of terminals and {antecedent, consequent, rule, rule_name, variable, object, list_of_lists, list, evaluation, string_list, string, nature, i} is the set of the nonterminals. The *charstring* terminal may be any sequence of alphanumeric characters or special symbols, the *stringname* terminal may be any sequence of alphanumeric characters or underscore character with the restriction that the first character must be a lower case letter, and the *integer* terminal ranges between 0 and n , $n \in \mathbb{N}$. The values of the evaluation nonterminal are depended on the rule that contains the evaluation nonterminal. These rules and the assigned values for the evaluation nonterminal will be described in the following.

```
test_object(Object, List_of_realisable_conditions,
            List_of_not_realisable_conditions,
            List_of_conditions_which_have_been_realised)
```

This rule infers what has to be done with reference to be an object started or what is necessary with reference to be an object terminated.

```
start_group_of_activities(Group, Evaluation,
                        List_of_realisable_activities,
                        List_of_not_realisable_activities,
                        List_of_activities_that_have_been_realised)
```

This rule makes inferences about the status of a group of activities which means, in what depth of the group of activities has the process proceeded in. It returns the Evaluation; the permitted values are:

- 'success' if the list of realisable activities is not empty or the list of not realisable activities is empty; in this case, that is if the group of activities is allowed to start, the *state* of the group of activities is set to 'started'
- 'failure' if the list of realisable activities is empty and the list of not realisable activities is not empty
- 'already started' if the *state* of group of activities is started
- 'already terminated' if the *state* of group of activities is terminated.

```
do_activity(Activity, Evaluation,
```

Table II. The rule syntax in BNF form

```
rule → if antecedent then consequent
antecedent → rule-name ( variable )
rule-name → stringname
variable → object , list_of_lists |
           object , evaluation , list_of_lists
object → string
list_of_lists → list |
              list, list_of_lists
list → [string_list]
string_list → string |
            string, string_list
string → charstring
evaluation → success | failure | failure in the context |
            failure out of the context | already started |
            not started | already terminated
consequent → (nature, string)
nature → ARTIFACT | ACTIVITY | LEVEL i
i → integer
```



```
List_of_realisable_artifacts,
List_of_not_realisable_artifacts,
List_of_artifacts_which_have_been_realised,
List_of_the_started_contexts,
List_of_the_not_started_contexts).
```

This rule makes inferences about the permission of an activity to be done. It returns the *Evaluation*. It returns, also, the contexts that the activity is realised; context is the group of activities within which the activity is realised. The permitted values are:

- 'success' if the list of started contexts is not empty and the list of not realisable artifacts is empty; the *state* of activity is set to 'terminated' and the *state* of the produced artifacts is set to 'existing'
- 'failure out of the context' if the list of started contexts is empty
- 'failure in the context' if the list of not realisable artifacts and the list of started contexts are not empty
- 'already started' if the *state* of the activity is 'started'
- 'already terminated' if the *state* of the activity is 'terminated'.

```
search_context(Activity,
List_of_started_group,
List_of_not_started_or_terminated_groups).
```

This rule makes inferences about the location of an activity. It searches the groups of activities and returns the one that the activity is found in.

```
terminate_group_of_activities(Group, Evaluation,
List_of_realisable_activities,
List_of_not_realisable_activities,
List_of_activities_that_have_been_realised).
```

This rule makes inferences about the validity of the termination of a group of activities. It returns *Evaluation*; the permitted values are:

- 'success' if the list of realisable activities and the list of not realisable activities are empty and the *state* of the group of activities is 'started'; the *state* of the group of activities is set to 'terminated'
- 'failure' if the list of realisable activities or the list of non-realizable activities are not empty and the *state* of the group of activities is 'started'
- 'already terminated' if the *state* of the group of activities is 'terminated'
- 'not started' if the *state* of the group of activities is 'not started'.

The Tutoring module transmits the trainees' queries to the Methodology module, filters its messages, according to the initial options and the trainees' actions, and accompanies them with the suitable messages. It uses the Help Knowledge Base, that con-

tains predefined dialogues, for its verbalism. Accordingly to the advancement of the tutoring process and the user model that HMeT keeps for every trainee, Tutoring module adapts its dialogues. An adaptation mechanism uses the produced and the needed artifacts in order to explain how the methodology proceeds or what actions are necessary for the methodology continuation. The predefined dialogues and the produced data during the tutoring process are composed by the Tutoring module and are presented to the trainees. The parameters used by Tutoring module are of two kinds: those that correspond to fixed options in the commencement of the simulation or the runtime process and may be changed by the authors and those that depend on the trainees' performance. The former are those that determine the educational strategy that will be followed, the maximum number of errors and the help level. During simulation of the methodology, the Tutoring module judges the correctness of a trainee's action and displays the appropriate messages. The maximum number of errors is a positive integer which, when reached, the process is stopped (if it is 0 there is no limit on the number of errors). The help level is an integer between 0 and 3 in case of failure or 0 and 1 in case of success. The larger the number of the help level the more the help that Tutoring module outputs. In simulation the help level is always equal to 0, otherwise it depends on the number of trainees' consecutive errors. The parameters that depend on the trainees' actions are the number of consecutive errors, the total number of the errors and the depth of help level. The number of consecutive errors is an integer between 0 and 4, which indicates the number of consecutive trainees' errors. The total number of errors is an integer between 0 and author-defined MAXERR, which indicates the total allowed number of trainees' errors along the runtime process. The depth of help level is an integer calculated by Tutoring module and it depends on the educational strategy and the number of consecutive trainees' errors.

Conclusions

In this paper, a hybrid expert system (HMeT) that has been used as embedded module in a ITS generator (GENITOR), was presented. The hybrid expert system was used in order to relieve the authors from having to anticipate all the possible (correct and incorrect) sequences or combinations of methodology elements that trainees may form, and then provide system responses for each. Authors simply describe these elements; it is the task of HMeT to ensure their valid ordering and to mediate tutoring interaction. It is estimated that average authors will use less than five inheritance levels and a relatively small number of

frame instantiations. Thus, the average methodology structure should not impose heavy computational overheads. The simplicity of the authoring process and the efficient and effective manipulation of the knowledge base made the hybrid scheme preferable. The authoring simplicity has been achieved thanks to the knowledge representation formalism that the system employs. HMeT utilises its own graphical environments for knowledge base construction and tutoring process and supports fast application development, prototyping and validation through simulation.

The whole development process of HMeT took about a person year. Two development platforms were used for HMeT implementation; the Authoring Interface, the Tutoring Interface and the Advancement Graphical Representation module were implemented in Borland C++ 4.5, whilst the Authoring Module, the Methodology Module and the Tutoring Module were implemented in Amzi Prolog 3.3. The whole integrated application runs under Windows 3.1 or later. During design, great care was taken to facilitate the interaction between system and authors. In general, the authoring process is rather complex, entailing several user actions of diverse context. To face this problem, the system adopts an interaction metaphor based on a goal-plan decomposition of the authoring process and clearly represents context of operation and authoring actions. The development of a methodology using HMeT has been dramatically simplified comparatively to the same process using MeT. The Authoring Module undertakes the knowledge base generation, creates the relations between the objects and prevents the invalid links; the authors just specify the names of the objects that will be used by the system. However, in order for a methodology to be a complete tutoring application, it also requires the construction of the declarative knowledge, the creation of exercises and tests etc. All the appropriate tools are provided by GENITOR. The testing and validation of the methodology have been simplified, too. The graphical user interface used by HMeT contains all the necessary information for the realisation of this process without any information overhead.

HMeT was validated in two phases. The first concerned HMeT as a stand alone system whilst the second concerned HMeT as an embedded module within GENITOR and involved the redevelopment of an intelligent training application. This application was METHODMAN II+, an ITS which aims at teaching the adaptation of MEDOC methodology in the domain of software project management. The development of the methodology using HMeT took a non-expert author less than half the time it has taken using MeT which was approximately one person week.

Future plans include the design and development of a large society of specialized agents who will implement several tutoring strategies by forming subject- or strategy-dependent teams. In the context of GENITOR, the authors of this paper aim to investigate the efficiency of hierarchical and non-hierarchical formation of agent teams. Furthermore, the current system version has to be supplemented with intelligent modules for multimedia database management, user interactions management and methodology specification, in order for GENITOR to be able to handle generic methodologies that will be described through a user-friendly interface and presented in full multimedia.

References

- Anderson, J.R., Boyle, C.F., Corbett, A.T. and Lewis, M.W. (1990).
'Cognitive Modelling and Intelligent Tutoring', in W.J. Clancey and E. Soloway, (eds), *Artificial Intelligence and Learning Environments*, 7-49, MIT Press.
- Fikes, R. and Kehler T. (1985).
'The Role of Frame-based representation in Reasoning'. *Communications of the ACM*, 28(9), 904-920.
- Gonzalez, A.J. and Dankel, D.D. (1993).
The Engineering of Knowledge-based Systems, Prentice-Hall, Inc.
- Graham, I. and Jones, P. L. (1988).
Expert Systems, Knowledge, Uncertainty and Decision. Chapman and Hall Computing.
- Jackson, P. (1986).
Introduction to Expert Systems. Addison-Wesley Publishing Company, Inc.
- Kameas, A.D. and Pintelas, P.E. (1997).
The Functional Architecture and Interaction Model of a GENERator of Intelligent TutORing applications, *Journal on Systems and Software* Elsevier Science Inc (to appear).
- Lucas, P. and van der Gaag, L.C. (1991).
Principles of Expert Systems, Addison-Wesley.
- Minsky, M. (1975).
'A Framework for Representing Knowledge', in P. H. Winston, (ed.), *Psychology of Computer Vision*, Cambridge, MA: MIT Press.
- Mispelkamp, H. (1992).
Generic tools for courseware authoring, in S. A. Cerri and J. Whiting, (eds.), *Learning Technology in the European Communities*, 585-593, Kluwer Academic Publishers.
- Quillian, M.R. (1968).
Semantic Memory. In Minsky, M. (ed.), *Semantic Information Processing*, Cambridge, MA: MIT Press.

Rickel, J.W. (1989).

Intelligent Computer-Aided Instruction: a survey organized around system components, *IEEE*

Transactions on Systems, Man and Cybernetics, 19(1): 40-57.

Schank, R.C. and Abelson R.P. (1977).

Scripts, Plans, Goals and Understanding. Hillsdale, NJ: Lawrence Erlbaum.

Stefik, M.J. and Bobrow, D.G. (1984).

'Object-oriented programming: themes and variations'. *The AI Magazine*, 2(4): 40-62.

Waterman, D.A. (1986).

A Guide to Expert Systems, Addison-Wesley Publishing Company, Inc.

Woolf, B.P. (1987).

Theoretical frontiers in building a machine tutor, in P. Kearsley, (ed.), *Artificial Intelligence and Instruction*, 229-267, Addison-Wesley.

Zaharakis, I.D., Kameas, A.D. and Pintelas, P.E. (1994).

MeT: The Expert Methodology Tutor of GENITOR, *Microprocessing and Microprogramming*, 40(10-12): 855-860.