Submitted
Research Papers

Acquiring Solved Problem Cases in a Training System Shell

XUEMING HUANG

Institute for Information Technology
National Research Council Canada
Ottawa, Ontario, Canada K1A 0R6

Abstract: Apprenticeship-based training systems have been built by AI researchers, but knowledge acquisition for the training systems remains a difficult problem. To tackle this problem, we developed a case acquisition system (CAS) for a training system shell (TRAISS). CAS helps a domain expert to enter a solved problem case into the system. It uses a dependency maintenance technique to convert the expert's solution into a multiple path directed graph that is consistent with a problem solving methodology in the workplace. Once the case is acquired, the tutoring module in TRAISS can provide a tutorial, based on the directed graph, for a trainee, using an apprenticeship-based approach.

1. Introduction

There has been interest among AI researchers in building apprenticeship-based training systems for the workplace (Brahan, Farley, Orchard, Parent & Phan, 1992; Lajoie & Lesgold, 1989; Newman, 1989). Such training systems provide tutorials based on a set of solved problem cases. Each case is a solution space for a problem in the application domain. The apprenticeship-based approach allows the trainee to learn concepts and skills directly related to the tasks in the workplace. However, knowledge acquisition is difficult for developing such a training system, similar to the situation for other types of tutoring systems (Clancey & Joerger, 1988). Constructing the case library requires a large amount of knowledge engineering work. In addition, a fixed set of cases usually does not meet the needs of the dynamic environment in the workplace. The initial set of pre-stored problem cases designed by the knowledge engineers may become out of date. However, without expertise in knowledge engineering and knowledge about the design of the system, it is difficult, if not impossible, for someone in the workplace to install a new problem case into the system. Consequently, the system may soon become obsolete in a changing user environment.

To overcome this difficulty, we have developed a training system shell, *TRAISS*, for the workplace. TRAISS works in two modes. In the knowledge acquisition mode, It supports a domain expert to enter solved problem cases. To enter a case, the expert need only solve the problem using an interactive computer-based tool, as he/she does in the day-to-day work, and

respond to prompts that ask for the reasoning of the action taken at each step of his/her solution. TRAISS uses a set of domain rules to generate a dependency graph of the expert's solution. Based on the dependency graph, it converts the expert's solution into a directed graph that contains multiple paths toward the solution. In the tutoring mode, TRAISS guides a trainee through the solution of the problem, employing an apprenticeship-based tutoring strategy. The directed graph generated from the expert's solution is used to validate the trainee's solution and to demonstrate solution steps to the trainee. The reasoning is tailored to generate messages that help the trainee to understand the problem solving steps. Note that generalizing the expert's solution to a directed graph allows the trainee to use a different path from the expert's path to reach the solution. Without this generalization, an acquired case would have little flexibility and would be of limited use. This paper focuses on the case acquisition system (*CAS*) of TRAISS, but a brief description of the tutoring system is also provided, primarily to show how an acquired case is used in tutoring.

TRAISS is developed within the context of an intelligent Advisor that is intended to provide support for a user of a computer-based design tool to learn skills in the application domain (Brahan, Farley, Orchard, Parent & Phan, 1992). In the development of the Advisor, the conceptual modelling phase of database design was chosen as the first application domain. The Advisor helps the user in three ways. In a reactive mode, the Advisor responds to a user's question with an answer that relates to the user's immediate need to get on with the task at hand. In a proactive mode, the Advisor observes the user's interaction with the system and offers advice as appropriate. TRAISS plays the third role of the Advisor. In an apprenticeship-based approach, TRAISS uses pre-solved design cases and builds on the reactive and proactive functionalities to introduce the user to the domain concepts and design methodology.

Other ITS shells have been built. Some of them focus on the issue of instructional planning (Macmillan & Sleeman, 1987; Peachey & McCalla, 1986). The others provide tools for authoring curricula, teaching strategies and domain knowledge (Chan, 1992; Clancey & Joerger, 1988; Murray & Woolf, 1992). However, these systems have been limited to development of more or less conventional knowledge bases with little attention given to support for creation and maintenance of libraries of solved problem cases required for an apprenticeship-based training system. TRAISS is intended to meet this requirement. It provides a convenient mechanism, CAS, for the domain expert to produce tutoring cases without intervention of a knowledge engineer. A unique contribution of CAS is the introduction of the dependency graph technique to knowledge acquisition in a training system. This technique allows the system to generate a generalized design solution necessary to support an apprenticeship-based approach to training.

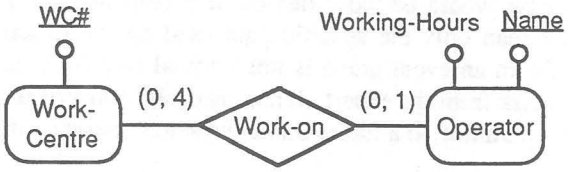
The rest of the paper is organized as follows. Section 2 discusses the knowledge representation for cases in TRAISS. Section 3 describes the architecture and the algorithm of CAS. Section 4 briefly discusses the tutoring system. Section 5 concludes the paper.

2. Representation of Solved Problem Cases

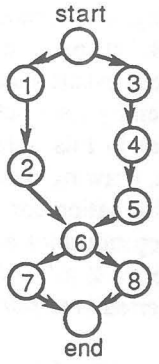
A problem may be solved in many different ways. In addition, there may be many ways to produce erroneous solutions. Much ITS work has been devoted to modelling various (correct and incorrect) ways in which learners produce problem solutions (Wenger, 1987). TRAISS is not intended to solve this classic ITS problem. Instead, TRAISS uses an apprenticeship-based tutoring strategy to guide the trainee through a particular methodology of solving a problem. For example, TRAISS might use such a simple methodology to solve a data modelling problem: divide the problem into small subproblems; for each subproblem, first create the entity classes and some basic attributes of the entity classes; then create relationship classes to connect the created entity classes and add more attributes to the entity classes if necessary; then set the cardinality pairs for

the links between the entity classes and the relationship classes. The trainee is not allowed to diverge from this methodology, no matter whether his/her divergent action eventually leads to a correct solution or not. The hypothesis behind this strategy is that only after the trainee has learned to apply one particular design methodology, will he/she have acquired the knowledge necessary to assess other methodologies for solving similar problems.

Although the trainee is restricted to working with one specific methodology, he/she may use any of the paths that are consistent with this methodology to arrive at the solution. For any given case, there may exist many such paths. To capture these paths, we use an *AND graph* (an AND/OR graph (Rich, 1983) that has only AND branches) to represent a case. Each node in the graph is called an *event* of the case. An event contains a problem solving action, a brief description of the action, the reason why the action is chosen in the particular situation. (For convenience, we refer to the two types of information other than the problem solving action as the *support information* of an event.) Links in the graph indicate the sequences of event exploration permitted by the system. An event can be explored only after all its parents have been completed. In each case, there exists exactly one *start event*, the eldest ancestor of all other events, and one *end event*, the youngest descendant of all other events. We call such a graph the *event graph* of the case. To complete a case, the student must complete every event in the graph and reach the end event.



(a) An ER model for a work centre situation



Operations:

1: (ce Work-Centre)
2: (ceid Work-Centre WC#)
3: (ce Operator)
4: (ceid Operator Name)
5: (cea Operator Working-Hours)
6: (cr Work-on Work-Centre Operator)
7: (sc Work-Centre Work-on 0 4)
8: (sc Operator Work-on 0 1)

(b) An event graph of the ER model

Figure 1 An event graph for an ER model

For example, Figure 1 (a) is an entity-relationship database model (*ER model*) (Chen, 1976) for a work centre situation. In this situation, a work centre can be operated by at most four operators and possibly none. An operator may be allocated to at most one work centre and

possibly none. A work centre is characterized by its number. An operator is characterized by a name and the number of working hours a day. Abbreviations have been used to name the operations. For example, *ce*, *ceid* and *sc* are used for create-entity, create-entity-id and set-cardinalities, respectively. The event graph in Figure 1 (b) contains ten events of constructing the ER model. With this graph, one may solve the problem by first creating the entity class *Work-Centre* and its identifier *WC#* (using operations 1 and 2) and then creating the entity class *Operator*, its identifier *Name* and its attribute *Working-Hours* (using operations 3, 4 and 5); or vice versa. After that one may create the relationship class *Work-on* between the two entity classes. Finally, one may set, in either order, the cardinality pairs for the two links of the relationship class. Note that it is possible to do operation 5 before operation 4, but good practice dictates that the identifiers should be created before other attributes. The event graph in Figure 1 (b) reflects this kind of expertise.

3. The Case Acquisition System

3.1. The Architecture

The *Case Acquisition System (CAS)* provides an interface for a domain expert to specify a solved problem case: a solution of the problem and a description of the expert's reasoning at each step of the solution. A case would be more flexible if it contains a multiplicity of commonly used "good paths" rather than only the specific path used by the expert. But specifying the branches of all these paths in an event graph is not a trivial task for a domain expert. CAS is designed to take over this task from the expert. It automatically converts the expert's solution into an event graph that allows a trainee, in a future training session, to use a different path to solve the problem.

The basic idea in CAS is the following: in an event graph, a link between two events actually represents either a *dependency* between the actions of the two events or a *preferred order* of carrying out the actions. These dependencies and preferred orders are governed by a set of domain rules (called *dependency rules*). For example, in ER modelling, one cannot create an identifier of an entity class before the entity class is created. Thus, the link between events 1 and 2 in Figure 1 (b) represents a dependency. On the other hand, the link between events 4 and 5 reflects a preferred order, in that domain experts usually create identifiers of an entity class before other attributes of the entity class, although it is not necessary to do the work in this order. CAS dynamically applies these rules when a domain expert solves a problem, drawing links between the events. When the solution is complete, these links contain the information for constructing an event graph. For convenience, we will generally refer to both dependencies and preferred orders as *dependencies*. If a distinction is needed, the former will be referred to as *hard dependencies*, for they compulsorily enforce action orders, while the latter will be referred to as *soft dependencies*, for they only recommend action orders.

Figure 2 shows the system architecture of CAS. CAS consists of four components: the *User Interface (UI)*, the *Dependency Recognizer (DR)*, the *Dependency Maintenance System (DMS)* and the *Event Graph Constructor (EGC)*. The UI provides a graphic drawing tool for a domain expert to enter an ER modelling solution. It also prompts the user to enter the *support information* for each event. Whenever the domain expert carries out an action, the UI identifies the action and passes it to the DR which then applies the dependency rules to draw dependencies between this action and previous actions. The action and the dependencies are passed to the DMS. The support information is temporarily stored in the UI. The DMS constructs a graph (called a *DMS graph*) to record the dependencies. It creates a DMS node for an event and a link for a dependency. The

DMS also provides a set of operations that allow the system to query and to revise the DMS graph. When the domain expert completes the solution, the EGC is invoked to convert the DMS graph into an event graph. The UI then graphically depicts the event graph. The domain expert is asked to confirm or to revise the event graph at this stage. If he/she confirms the graph, the UI would incorporate the support information into the case. This final case is then saved in the case library of TRAISS.

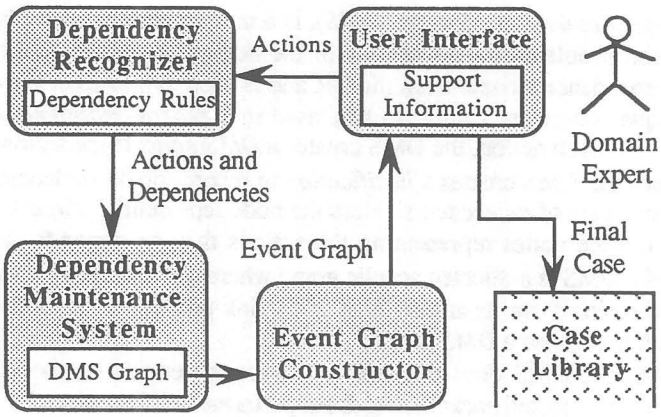


Figure 2 The Architecture of CAS

3.2. The Dependency Recognizer

The Dependency Recognizer (DR) is responsible for recognizing dependencies between an action just taken by the domain expert and previous actions in the current problem solution. The recognition is accomplished by applying relevant dependency rules in the DR's rule base. There are two types of dependency rules in the DR. *Hard dependency rules* describe how problem solving operations in a particular domain depend on each other. The hard dependencies generated from these rules are compulsory. The domain expert is not allowed to delete a hard dependency. In the domain of ER modelling, 15 hard dependency rules have been identified and encoded in the current system. Each of these rules describes the prerequisite of an ER modelling operation. For example, one of the rules is written as:

(CR rel ent-1 ent-2) <== ((CE ent-1) $\vee$ (RE ? ent-1)) $\wedge$ ((CE ent-2) $\vee$ (RE ? ent-2))

It says that at the time of creating a relationship class between two entity classes, ent-1 and ent-2, each of the two entity classes must already be either created or renamed as ent-1/ent-2 from another entity class. If the prerequisite is satisfied, dependencies are drawn between the actions.

Soft dependency rules are of the same form as hard dependency rules. However, rather than being based on basic properties of the domain, a soft dependency rule is based on a methodology used by domain experts. For example, a currently implemented soft dependency rule states that an attribute of an entity class should not be created before an identifier of the entity class unless the expert has created them in that order. The dependency rule base may contain several sets of soft dependency rules. Each set represents a particular methodology. When a domain expert selects a methodology, the set of soft dependency rules corresponding to the selected methodology is activated. However, a methodology in a workplace might be vague and applied differently in

different cases. The set of rules might not accurately describe it, so the system might generate a soft dependency inconsistent with the methodology in the application to a particular case. Thus, it is important to allow the domain expert to delete/add a soft dependency from/to the event graph when the case is revised.

3.3. The Dependency Maintenance System

The *Dependency Maintenance System* (*DMS*), is a mechanism that records and manipulates dependencies between problem solving actions in the domain expert's solution. In CAS, this information about dependencies come from the DR and is used to construct the event graph. The DMS uses a technique somewhat similar to that used in a *reason maintenance system (RMS)* [McAllester, 1990]. For each action, the DMS creates a *DMS node*. If the action, say A_0, depends on some other actions, the DMS creates a *justification* to record the dependencies. A justification has a *consequence* and a set of *antecedents*. Here the node representing A_0 is the consequence of the justification, while the nodes representing the actions that A_0 depends on are antecedents. Thus, the database of a DMS is a directed acyclic graph where, for each justification, there are links pointing to the justification from its antecedents and a link pointing to its consequence from the justification. We call this graph a *DMS graph*.

Unlike an RMS, the DMS does not deal with inconsistencies --- inconsistencies are not recorded, and dependency-directed backtracking is not performed. We assume that a domain expert's solution is correct and consistent. On the other hand, to allow the domain expert to revise the event graph, the DMS provides operations to revise and to delete a justification. These kinds of operations are normally not provided by an RMS [Doyle, 1979; de Kleer, 1986]. Also, the DMS distinguishes *hard antecedents* and *soft antecedents* of a justification. If the dependency between an antecedent and the consequence of the justification is a hard dependency, then the antecedent is a hard antecedent. If the dependency between them is a soft dependency (a preferred order), then the antecedent is a soft antecedent. Encoding this kind of distinction gives CAS the ability to prevent removal of a hard dependency when the domain expert later revises the event graph (see Section 3.5). It also allows the Tutor that later uses the event graph to provide more informative advice to the trainee. For example, if the trainee's actions had violated a hard dependency (he/she took the consequence action before a hard antecedent action), then the Tutor could say, "You cannot do so because ....". But if the trainee's actions had only violated a soft dependency, the Tutor would say, "You could do so, but with the methodology we are using, you should ....".[1]

Table 1 shows a list of DMS operations. These operations allow the user (in CAS the user is the Event Graph Constructor) to record, to query and to manipulate dependencies between actions in a domain expert's solution. Note that in the explanations of the two query operations, **Depend?** and **Hard-depend?**, the concept *"depends on"* is defined to be a transitive relation between nodes. That is, for nodes n_1, n_2 and n_3, if n_1 depends on n_2 and n_2 depends on n_3, then n_1 depends on n_3.

---

1 At the current development stage, this information has not been used to tailor advice to the trainee.

Table 1: The DMS operations

Create-dg(name): Create a DMS graph with the provided *name*.

Create-dg-node(dg): Create a node for the DMS graph *dg*.

Add-justification(rule, consequence, hard-antecedents, soft-antecedents): Create a justification which has the indicated *consequence*, *hard-antecedents* and *soft-antecedents*. The argument *rule* indicates the dependency rule based on which the justification is drawn.

Delete-justification(justification): Delete a *justification*.

Depend?(n_1, n_2): Return t (true) if node n_1 depends on, via hard dependencies or soft dependencies, node n_2; otherwise return nil (false).

Hard-depend?(n_1, n_2): Return t (true) if node n_1 depends on, via only hard dependencies, node n_2; otherwise return nil (false).

Add-hard-antecedents(justification, new-hard-ants): Add a set of hard antecedents *new-hard-ants* to the indicated *justification*.

Delete-hard-antecedents(justification, old-hard-ants): Delete a set of hard antecedents *old-hard-ants* from the indicated *justification*.

Add-soft-antecedents(justification, new-soft-ants): Add a set of soft antecedents *new-soft-ants* to the indicated *justification*.

Delete-soft-antecedents(justification, old-soft-ants): Delete a set of soft antecedents *old-soft-ants* from the indicated *justification*.

3.4. The Event Graph Constructor

When the domain expert enters the complete solution, the Event Graph Constructor (EGC) is invoked to convert the DMS graph to an event graph. Basically, the EGC removes redundant links between the nodes. It also adds a start event and an end event to the graph. A link is redundant if after removing the link, the parent event on the link still remains as an *ancestor* of the child event on the link (an event e_1 is an ancestor of another event e_2 if e_1 is a parent of e_2 or if e_1 is an ancestor of a parent of e_2). For example, the link between event 1 and event 3 in Figure 3 is redundant because 3 can be executed only after both 1 and 2 are completed, whether the link exists or not. The following is the EGC algorithm. Obviously, it is polynomial in the number of nodes of the DMS graph.

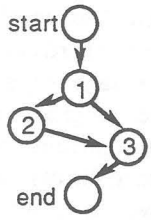


Figure 3 An event graph with a redundant link

Algorithm EGC (DG) {The algorithm generates an event graph EG}

(1) Construct a temporary event graph EG' from the DMS graph DG in the following way:
   * Take the antecedents of each justification as parents of the consequence of the justification;
   * Create a start event as the parent of the nodes that presently have no parent.
   * Create an end event as the child of the nodes that presently have no child.

(2) For each node n ∈ EG', do {remove redundant links from EG'}
   Let S be the set of parents of n and for each p ∈ S, do
   * Check if there is another p' ∈ S such that p' depends on p (use the operation **Depend?**(p', p));
   * If there is such a p', delete p from S.

{end of EGC}

The following function implements the DMS operation **Depend?** used in EGC during step (2).

Function **Depend?** (n1, n2)

(1) For each parent p of n_1 without a "visited" mark, do
   If p = n_2, then return *true*, otherwise
   * Mark p "visited";
   * Call **Depend?** (p, n_2);

(2) Return *false*.

3.5. The User Interface

The User Interface (UI) interacts with the domain expert to acquire the solved problem case. The UI contains a *Graphic Drawing Tool*, an *Interactive Window*, a *Graph Display Window* and an *Event Entry Editor*. The Graphic Drawing Tool is the tool used in the day-to-day design activities in the workplace. It is used both by the domain expert who creates the problem case and by the trainee who is introduced to solve the case. As the expert uses the Graphic Drawing Tool to solve the design problem, the UI recognizes the expert's actions and passes them to the DR. Also, through the Interactive Window the system prompts the expert to enter the support information for each event.

When the case is completed, the UI invokes the Graph Display Window to display the event graph constructed by the EGC. It also prompts the expert to confirm or to revise the graph. A simple operational language is provided for the revision. The operation language provides two types of operations: (before ?e1 ?e2) and (parallel ?e1 ?e2), where ?e1 and ?e2 are variables of events. For example, if (before event-1 event-2) is stated, then a soft dependency is added between event-1 and event-2. If (parallel event-3 event-4) is stated, then the soft dependency between event-3 and event-4 is removed. However, if there is a hard dependency between event-3 and event-4, the operation is not carried out and an error message is given. When the expert finishes the revision, the EGC is invoked again to create a new event graph based on the revised DMS graph. Then the UI displays the new graph. If the expert confirms the graph, the UI incorporates the support information into the case and saves the case to the case library.

The Event Entry Editor provides a way to revise the support information. It is particularly useful because the linearity of the problem solving process may have impact on the expert's

descriptions of the situations and the reasonings. The Editor allows the expert, when seeing the event graph, to revise the descriptions so that they suit the non-linearity of the graph.

4. The Tutor

The Tutor uses apprenticeship learning as the basic model (Collins, Brown & Norman, 1987). This approach emphasizes the acquisition of skills directly related to the accomplishment of meaningful tasks. Abstract concepts are only introduced as they are required to explain a given task. The trainee is guided through a set of examples that illustrate applications of the target skill set. Each of the examples is a problem case that a domain expert has entered using CAS.

During the process that the trainee solves a design problem, the Tutor watches the trainee who carries out problem solving actions. According to the event graph of the case, at each point of the problem solving, there are several actions that may be carried out. If the trainee's action matches one of these possible actions, then the trainee's action is consistent with the design methodology used by the domain expert who constructed the case. Otherwise, the trainee's action diverges from the methodology. In this case, the Tutor gives a warning to the trainee, pointing out the divergence and putting the trainee into the help mode. Also, at any time that the trainee has difficulty continuing, he/she can ask for help from the Tutor. Several types of help are available: (1) listing the possible actions of the same type as the trainee's action (e.g., creating an entity); (2) listing all possible actions; (3) explaining why a possible action is suitable at that particular time; and (4) showing how to carry out a possible action. The help messages are generated using support information and the event graph of the case.

The Tutor has a *User Model* that contains a record of the trainee's knowledge level with respect to each concept and each type of task in the application domain (e.g., to create an entity class). The knowledge levels are estimated based on what the trainee has done and what the system has shown to the trainee in the past interactions. Whenever the trainee asks the system to show him/her how to do a task, the system consults the User Model and decides whether a hint should be given first or a demonstration should be carried out. A current research activity is to investigate the issue of how to use knowledge in the User Model to select problem case for the trainee and to tailor the help messages to suit the individual trainee.

Note that the Tutor is embedded in the Advisor environment that gives the trainee full access to other modes of the Advisor. At any time, the trainee can enter the Reactive Help mode to ask questions about concepts in the domain. The Reactive Help module applies case-independent domain knowledge in the Advisor's knowledge base to generate answers for the questions. Also, throughout the tutorial session, the Proactive Help module applies this domain knowledge, "watching" for conceptual errors and providing help whenever appropriate. Details of these two modules of the Advisor is provided in (Brahan, Farley, Orchard, Parent & Phan, 1992).

5. Conclusion

We have presented a knowledge acquisition system, CAS, that acquires solved problem cases from the domain expert in the context of a training system shell. It allows the training program to be kept up-to-date by the end user organization without the need for in-house knowledge engineering, nor intervention from the designer of the training system. CAS uses dependency graph techniques in knowledge acquisition to provide the flexible case representation necessary to support an apprenticeship-based approach to training.

The first prototype of CAS has been implemented in Common Lisp on a Sun SPARCstation. Currently, only limited evaluation has been done, but the preliminary results are

encouraging. System functionality has been validated using a number of cases. The largest one has 15 entity classes and 19 relationship classes. A comparison with manual techniques has been made by re-entering a database design problem solution that had been generated previously by a database expert and subsequently generalized manually by a knowledge engineer. In each of these activities, results have indicated that the system offers potential for higher productivity and for more consistent and error-free cases than can be produced by the knowledge engineer alone.

Several topics remain to be investigated. The current system deals only with AND graphs. That is, a problem is associated only with one case. If there are multiple cases for a problem, then an AND/OR graph (Rich, 1983) must be used. But then the tutoring system would be required to figure out which case the trainee follows, as what is required in a conventional ITS (Wenger, 1987). This would also require to develop an algorithm to merge several cases derived from the expert's different solutions into a single AND/OR graph. The current system provides no facilities for managing the case library. In addition to identifying a hierarchy of levels of difficulty and an indexing system that relates cases to introduction of specific concepts and tasks, there is the potential for using parts of cases to illustrate design procedures in the tutorial interaction and in the other components of the Advisor. It is also possible to use a goal-plan structure to represent each primitive component of a case, as what is done in the Lisp Tutor (Anderson & Brian, 1985). To deal with the variation and the dynamics of problem solving methodologies in the workplace, it is also necessary to provide facilities for a domain expert to define a methodology (i.e., a set of soft dependency rules) and to activate it later.

Acknowledgements

Thanks to J. W. Brahan and Sieu Phan for their valuable inputs to this research. François Vernadat and Luen Law used CAS to develop design cases. Philippe Davidson implemented an early version of the tutor. Gordon McCalla, Peter Clark, Peter Turney and Martin Brooks read the paper and gave valuable comments.

References

Anderson, J. R. (1985). The Lisp Tutor. BYTE, 197-207.

Brahan, J. W., Farley, B., Orchard, R. A., Parent, A., Phan, C. S. (1992). A designer's consultant. In: Bramer, M. A. and Milne, R. W. (eds.), *Research and Development in Expert Systems IX*, Cambridge University Press, 197-207.

Chan, T. (1992). Curriculum tree: a knowledge-based architecture for intelligent tutoring system. *Proceedings of the Second International Conference on Intelligent Tutoring Systems (ITS '92)*, Montreal, Canada, 140-147.

Chen, P. P. (1976). The entity-relationship model --- toward a unified view of data. *ACM Transactions on Database Systems* 1, 9-36.

Clancey, W. J. and Joerger, K. (1988). A practical authoring shell for apprenticeship learning. *Proceedings of the International Conference on Intelligent Tutoring Systems (ITS '88)*, Montreal, Canada, 67-74.

Collins, A., Brown, J. S. and Newman, S. (1987). *Cognitive Apprenticeship: Teaching the Craft of Reading, Writing and Mathematics*, BBN Technical Report 403.

de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence* 28, 127-162.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence* 12, 231-272.

Lajoie, S. P. and Lesgold, A. (1989). Apprenticeship training in the workplace: computer-coached practice environment as a new form of apprenticeship", *Machine-Mediated Learning* 3, 7-28.

Macmillan, S. A. and Sleeman, D. H. (1987). An architecture for self-improving instructional planner for intelligent tutoring system. *Computational Intelligence* 3, 17-27.

McAllester, D. A. (1990). Truth maintenance. *Proceedings AAAI-90*, Boston, 1109-1115.

Murray, T. and Woolf, B. P. (1992). Results of encoding knowledge with tutor construction tools. *Proceedings AAAI-92*, San Jose, California, 17-23.

Newman, D. (1989). Is a student model necessary? apprenticeship as a model for ITS. *Proceedings of the 4th International Conference on AI and Education*, 177-184, Amsterdam.

Peachey, D. R. and McCalla, G. I. (1986). Using planning techniques in intelligent tutoring systems. *International Journal of Man-Machine Studies* 24, 77-98.

Rich, E. (1983). *Artificial Intelligence*, McGraw-Hill, Inc.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems*, Morgan Kaufman Publishers, Inc.