Springer
Berlin
Heidelberg
New York
Barcelona
Budapest
Hongkong
London
Mailand
Paris
Santa Clara
Singapur

Friedbert Huber-Wäschle Helmut Schauer
Peter Widmayer (Hrsg.)

GISI 95

Herausforderungen eines globalen
Informationsverbundes für die Informatik

25. GI-Jahrestagung und
13. Schweizer Informatikertag
Zürich, 18.-20. September 1995

Springer

Towards Adaptive Learning Environments

Peter Brusilovsky, Marcus Specht, and Gerhard Weber
University of Trier
E-Mail: {plb | specht | weber}@cogpsy.uni-trier.de

Abstract: Existing intelligent learning environments for programming represent a step towards comprehensive adaptive learning environments that support all activities in learning programming. In most of these systems, however, only the tutoring component is adaptive. The user interface usually looks the same for the novice and for the advanced learner, while the student's knowledge of the subject matter strongly changes from the beginning to the end of a course. We argue that a next step towards adaptive learning environments is to make all its components adaptive. In this paper, we discuss some problems of creating adaptive environment components for intelligent learning environments and present our current work into this direction.

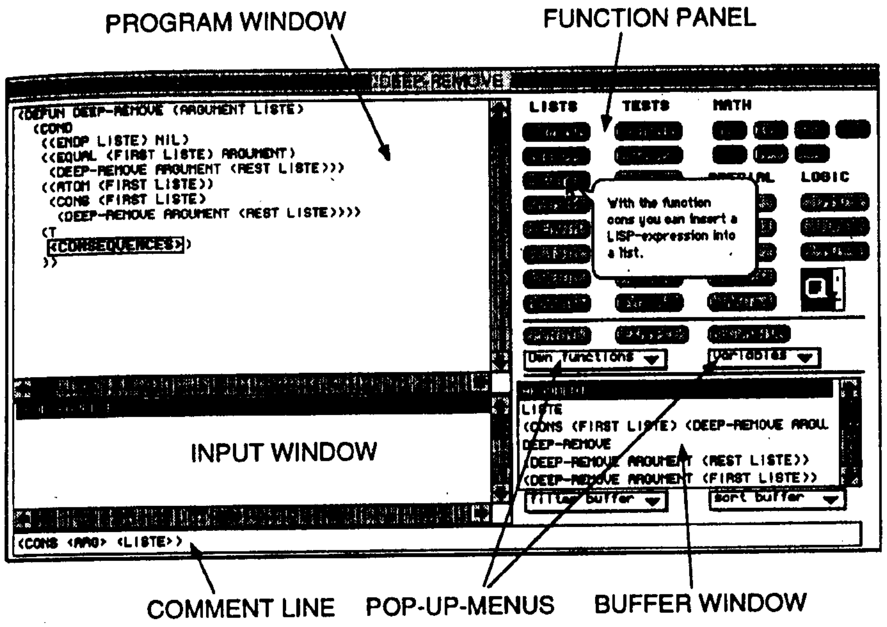
Introduction

Intelligent learning environments are a relatively new kind of intelligent educational systems. In addition to more traditional system-driven intelligent tutoring components that support students' *learning*, an intelligent learning environment (ILE) includes one or more student-driven components that support students' *doing*. These components form the environment.

A special case of an ILE with a complex and powerful environment component is an ILE for programming (here referred as Intelligent Programming Environment, IPE). An IPE contains both an intelligent coach or tutor to support student learning and a programming environment to support student programming activity. A number of IPEs have been described: APT (Corbett & Anderson, 1992), SYPROS (Herzog, 1992), DISCOVER (Ramadhan & du Boulay, 1993), ABSYNT (Möbus, Thole, & Schröder, 1993), GIL (Reiser, Kimberg, Lovett, & Ranney, 1992), ELM-PE (Weber & Möllenberg, 1995), and ITEM/IP (Brusilovsky, 1992).

Existing IPEs can be considered as the second step (after such classic ITSs as PROUST (Johnson, 1986) and Lisp Tutor (Anderson & Reiser, 1985)) towards "teachers' dream"—comprehensive adaptive learning environments to support all activities in learning programming. Most of these IPEs, however, are only partly adaptive. In most cases, only the tutoring component uses the student model for the purpose of adaptation. The user interface usually looks the same for the novice and for the advanced learner, while the student's knowledge of the subject matter strongly changes while learning. We argue that the next step towards an adaptive learning environment is to make all the components of an IPE adaptive.

In this paper, we discuss some problems of creating an adaptive interface for IPEs. We present our current work on the next version of ELM-PE where we expect to implement the idea of comprehensively adaptive IPE. In the following section, we briefly review the current stage of the ELM-PE project which forms the background of our current work. We describe the existing components that are most interesting in the scope of this paper. We then discuss the application of adaptive interface technology along with using the student model of the ILE as the user model for creating an adaptive interface for environment components. A particularly powerful student model is ELM—the episodic learner model used in ELM-PE. The possibilities of making an adaptive interface based on ELM are demonstrated at hand of ALFRED, the structure editor of ELM-PE. We consider some interface features of ALFRED and discuss how these features can be adapted using different types of knowledge about the student represented in ELM.



```
(DEFUN DEEP-REMOVE (ARGUMENT LISTE)
  (COND
   ((ENDP LISTE) NIL)
   ((EQUAL (FIRST LISTE) ARGUMENT)
    (DEEP-REMOVE ARGUMENT (REST LISTE)))
   ((ATOM (FIRST LISTE))
    (CONS (FIRST LISTE)
     (DEEP-REMOVE ARGUMENT (REST LISTE))))
   (T
    <CONSEQUENCES>)
   ))
```

```
LISTE
(CONS (FIRST LISTE) (DEEP-REMOVE ARGU
DEEP-REMOVE
(DEEP-REMOVE ARGUMENT (REST LISTE))
(DEEP-REMOVE ARGUMENT (FIRST LISTE))
```

```
(CONS <ARG> <LISTE>)
```

Figure 1: The syntax-driven structure editor ALFRED

Components of ELM-PE

The knowledge-based programming environment ELM-PE is designed to support novices learning the programming language LISP. It has several features which are especially useful in learning to solve problems in a new, complex domain. Some of these features will be sketched briefly in this section.

ALFRED—a syntax-driven structure editor. In order to reduce syntax errors, coding function definitions is supported by a syntax-driven LISP-editor. In the program window of the structure editor, program code can be produced by filling in slots of LISP expressions (Figure 1). These schemata can be accessed from buttons in the function panel or from typing in the first part of a function call. Additionally, expressions can be typed in directly via keyboard or can be pasted from the buffer window.

Example-based programming. Examples of LISP-code can be displayed in a separate example window. An example can be selected both by the user and by the system's analogy component from examples presented in the learning materials of the LISP-course as well as from the set of function definitions the student has already coded.

Visualization. One central principle in learning programming is to visualize the flow of data during evaluation of programs (e.g., Eisenstadt, Price, & Domingue, 1993). This is supported by a stepper showing all evaluation steps indicating corresponding expressions of self-defined function definitions—including definitions of sub-functions—in the program window. The stepper is supposed to help students envision dynamic program properties and to encourage active and self-directed learning. Additionally, those parts of the program code which are responsible for errors or are explained by the knowledge-based diagnosis can be highlighted in the program window.

Automatic cognitive diagnosis. The knowledge-based component of the programming environment consists of a cognitive diagnosis of the program code produced by a student. This diagnosis is based on ELM, a case-based learning model (Weber, 1994b). The diagnosis results in an explanation of how the program code could have been produced by the learner. This explanation is the basis for offering hints to the learner concerning which plans must be followed to solve subgoals during problem solving and what part of the code is incorrect.

Adaptive Interfaces

When learning in an ILE, a student spends most of the time working with the environment. Most existing educational environments provide the same interface for a novice and an experienced student, even though the student's knowledge of the subject strongly differs between the beginning and the end of the course. When the interface of the environment is oriented towards an experienced student, the interface appears to be too complex for a novice who does not get enough feedback and is often faced with unknown concepts and system features. When the interface is simple enough to be used by a novice, then it is usually restrictive and not powerful enough for a more advanced student. The problem is: how can the environment be constructed so that there is no threshold and no ceiling? It should be easy to get started, but these systems should also offer a rich functionality for experienced users (Fischer, 1988).

One promising approach suggested in (Brusilovsky, 1993) is to use the student model of the ILE as the user model for creating an adaptive interface applying techniques developed in the domain of adaptive interfaces (Dieterich, Malinowski, Kühme, & Schneider-Hufschmidt, 1993). This approach was implemented in ITEM/IP, an ILE for learning introductory programming (Brusilovsky, 1992). ITEM/IP applies a simple "conceptual" overlay student model to support both adaptive teaching and an adaptive interface. Using such a simple model the system can only reflect the level of user *conceptual knowledge*, i.e., knowledge about language elements (constructs, functions) and more high level programming concepts (plans, techniques). This knowledge is usually represented as a kind of conceptual network. As a result the methods of adaptation in ITEM/IP are relatively simple.

Generally speaking, in addition to conceptual knowledge the following student knowledge on the domain level can be taken into account for the purpose of adaptation: procedural knowledge and episodic knowledge. *Procedural knowledge* or skills represent the student's problem solving knowledge. This knowledge is usually represented as a set of goal-action or goal-plan rules and can be reflected by a procedural overlay student model or "buggy" model (e.g., the APT system, Corbett & Anderson, 1992). *Episodic knowledge* consists of cases describing problems and their solutions considered or solved by the student. An example of using and representing this kind of knowledge in the programming domain is provided by ELM-PE (Weber, 1994b).

We argue that all these kinds of knowledge are important for interface adaptation in IPE. The more kinds of student knowledge are represented in the student model, the more adaptation techniques can be used. Moreover, some advanced adaptation techniques require two or even all three kinds of knowledge to be represented. Unlike the simple conceptual overlay student model of ITEM/IP, the episodic learner model ELM is powerful enough to represent all three kinds of student knowledge on the domain level. The goal of our current research is to investigate the ways of using ELM to support making the interface components of ELM-PE adaptive. In the subsequent sections of this paper, we will present in more detail the episodic learner model of ELM-PE, and discuss some techniques that are used or can be used to adapt the ALFRED structure editor to various kinds of information about the student represented by ELM.

ELM: A Case-based Learning Model

ELM is a type of user or learner model storing knowledge about the user (learner) in terms of a collection of episodes. In the sense of *case-based learning*, such episodes can be viewed as cases. To construct the learner model, the code produced by a learner is analyzed in terms of the domain knowledge on the one side and to a task description on the other side. This cognitive diagnosis works according to explanation-based generalization (EBG) (Mitchell, Keller, & Kedar-Cabelli, 1986) and results in a derivation tree of concepts and rules the learner might have used to solve the problem. Concepts and rules contained in the derivation tree are instantiations of units from the knowledge base. The episodic learner model is made up of these instantiations (episodic frames) and generalizations based thereupon. A case in the episodic learner model—a stored interpreted episode—consists of a set of episodic frames that are distributed over the knowledge base.

On a first glance, ELM looks like an overlay model. Episodic instances represent information of how well a specific concept is mastered. However, episodic frames contain a lot of information that can be used for individual adaptation. They contain information about the episode and the context plan in which the concept was used, which rule applied in that situation, and store the corresponding part of the student's code.

Individual cases—or parts of them—can be used, up to now, in two different ways to adapt the system to a particular learner. On the one hand, episodic instances can be used during further analyses as shortcuts if the actual code and/or plan matches corresponding patterns in episodic instances (Weber, 1994b). On the other hand, cases can be used by an analogical component to show up similar examples and problems for reminding purposes (Weber, 1994a). In the following, we will show how information contained in ELM and other individual information is used and will be used to build an adaptive learning environment.

Knowledge-based Interface Adaptation in ALFRED

Adaptive vs. Adaptable Systems

When talking about adaptive systems, we have to take two different forms of adaptation into account: adaptability (e.g., supporting end-user modifiability) and adaptivity (e.g., a system adapts the interface itself using some data or knowledge about the user). A system aimed at supporting both novices and more advanced programmers has to be both adaptable and adaptive. While novices often need more guidance and want to be released from too complex tasks, advanced users want to have more control over the system. In advanced systems, therefore, the task of adaptation can be split up differently between user and system (Dieterich et al., 1993). From totally user-driven to exclusively system-driven systems every combination of adaptivity and adaptability is possible.

A simple but interesting form of adaptation is what we call *Last State* Adaptation. It means adaptation to the last state of the user-system interaction—the system just keeps the "settings" of the individual user (working directories, window positions, etc.) and comes up when starting like the last time the user worked with it. Despite the simplicity of this kind of adaptation, it is very useful and it is a shame that a lot of environments do not take it into account. In ALFRED, an example of last state adaptation is the permanently growing "own functions" pop-up-menu (Figure 1). When starting ELM-PE, the user finds all self-defined functions and variables in a menu and can use them as templates. Unlike adaptive techniques based on user models that are considered in the following sections, last state adaptation does not require any special modeling of knowledge about the user and does not require any decision-making. The topic of the following sections is how various kinds of knowledge about the user accumulated and processed by ELM can be used to make various interface components of the IPE adaptive.

Concept Level Adaptation: Function Panel

By concept level adaptation we mean a kind of adaptation which takes into account only the level of user knowledge on various domain concepts.

A good example of adaptation on the conceptual level of knowledge can be demonstrated by the function panel of ALFRED (Figure 1). A novice learning the basic concepts of LISP is confronted with a multitude of new expressions. Presenting all available constructs simultaneously (e.g., showing all functions in the function panel) can cause working memory overload to the novice. Brusilovsky (1993) suggested to use a conceptual overlay student model to hide all unlearned constructs from an editor's menu. In the ELM model, we have the information about the successful and erroneous usage of concepts and, therefore, can use several ways to adapt the panel to the student's individual knowledge about the function presented as a button. At present, we consider the following three techniques for function panel adaptation (which can be used in combination).

Hiding: Elements of the user interface which the student is not ready to use because he or she is lacking the prerequisite knowledge may be hidden from the screen.

Dimming. Recent research shows that stable layout of menus is important to some users. To keep the panel design the same regardless of the level of knowledge, irrelevant or too advanced function buttons could stay on the screen and just be dimmed. Dimming reduces cognitive overload and allows to build up a consistent model of the interface.

Outlining: Another way to adapt the panel would be to outline (using another color) the buttons for the function, which are new (newly learned) for the student, or learned not well enough. This is more an advice for the student to pay special attention to these functions.

When the above techniques are used, the interface's visibility and functionality incrementally grows along with the user's acquisition of knowledge. Such kind of adaptive interface can be called *incremental interface*. It is important that adaptation in the incremental interface should be made stepwise each time when some reasonably sized portion of new material (e.g., a chapter) is introduced or learned. Otherwise, too many changes in the interface can really upset the student.

Another technique of adaptation related to the function panel is adapting the "comment line" or the "balloon help" which are used to show the student some information about the function when he points to the button with the name of the function. Dependent on the students knowledge about the function the information presented can be varied. For the student that has no experience with a certain function, some introductory information (what the function is doing), for the advanced some more detailed information can be presented.

Episodic Level Adaptation: The Buffer

By episodic level adaptation we mean a kind of adaptation that takes into account the user's episodic knowledge. This knowledge can be effectively used for adaptation by ALFRED's buffer window (Figure 1). Students tend to develop a kind of individual style while learning programming. Using episodic knowledge, the system can determine which chunks (patterns) of code are most often and most successfully used by the student, which are used randomly, which are never used, and which often cause problems. Two interesting adaptation techniques can be considered.

Adaptive buffer filtering. The student working with ALFRED uses the buffer as a kind of "external memory" for storing some useful lisp expressions. Some of them appear only once while others are often re-used because they reflect a personal style and are specific to the problem. In many cases, the buffer helps the user to avoid retyping. However, it tends to grow too big and to become less helpful, because it is not so easy to find a required expression in a big scrollable buffer. Using ELM episodic knowledge, the system uses a caching technique that intelligently "filters" the current buffer and keeps those instances in the buffer, which the student usually prefers when coding.

Episodic buffer. Despite of using the buffer only to keep useful expressions collected during the current session, the system can provide the user with an option to load into the buffer the preferred student's clichés, which the system can collect and generalize over all previous sessions of work. On special request, even bigger chunks of code and also generalized plans can be provided in it.

Goal - Plan Level Adaptation: Example Library

By goal-plan level adaptation we mean a kind of adaptation that takes into account user procedural problem-solving knowledge. This knowledge is usually represented in form of goal-action or goal-plan rules (Anderson, 1993) and the student model usually reflects the probability of particular rules being acquired. This knowledge can easily be deduced from ELM and can be used for adaptation of various interface features. We demonstrate how this knowledge can be used for an example library in the ELM-PE interface. To support example-based programming, ELM-PE collects and stores all program examples presented to the student and solved by the student. Pushing the example button on the function panel, the system brings up the window with a list of all existing examples grouped into lessons. The student can choose any example. It will be shown to him or her in an editor window. The student can edit the code and insert any part of it into the program window.

The most suitable technology to adapt such an interface to the user goal-plan knowledge is *adaptive prompting* (Kühme, 1993). Knowing the user's general goal (i.e., the problem that the user is currently solving) and the goal-plan knowledge, the system can deduce the preferred way of solving the problem and thus determine which tools (here—plans and elementary functions) the user is most likely to use in problem solving. These tools can be made more easily accessible to the user. In case of the example library, the system can determine a subset of examples containing useful plans and chunks, which can be re-used in solving the current problem. Based on the episodic learner model, these examples can individually be selected as described in the next section. Names of these examples can be highlighted in the menu of examples. When one of these examples is selected a part of it containing the relevant piece of code can be outlined.

The adaptive prompting technology is an example of using interface adaptation to help the student in problem solving. It can be used not only in an example library interface, but also to highlight relevant buttons on the function panel or relevant lines in the buffer. The only precondition for the system to use this knowledge is to know which problem the student is currently solving. In contrast to the two previously considered technologies of adaptation, adaptive prompting can hardly be used when the student is solving an own problem or just playing with the editor.

Combining Goal - Plan and Episodic Level Knowledge: Best Example Selection

Adapting a programming environment to the learner's programming goals is one of the intelligent features that are very difficult to establish but that seem to be very promising to help individual users. Using adaptive prompting with an example library is just the most simple example of it. A more advanced case is the ability to select the most relevant example or reminding. This is already implemented in ELM-PE.

When working at a programming task to solve an exercise, the learner can select an example or reminding from the library. But, it may be difficult for a novice, even with the help of adaptive prompting, to decide what the best example will be that will help to solve the current problem. Therefore, a special button is provided for the learner to ask the system to show up the best suited example in the example window. Providing examples by the system is done by the explanation-based retrieval (EBR) method (Weber, 1994a). It is based on the episodic learner model ELM and requires both goal–plan and episodic level knowledge.

EBR searches for a best examples in three steps. First, an expected solution to the current problem is generated starting with a plan description of the current task preferring rules being used by the learner in previous tasks. This predicted solution is temporarily stored in the episodic learner model according to the process described earlier. The previous case whose episodic frames are stored in nearest distance to the temporarily stored frames is selected as the best examples at hand of an organizational similarity measure (Weber, 1994a).

Complex Adaptation: Example Explanation

Even if a user is provided with an appropriate example, it may be necessary to explain to him or her how the problems are related and what the differences are. Explanations of examples have a positive learning effect, especially if learning is accompanied by self-generated explanations. Thus, using examples may be improved by either forcing learners to build self-explanations, or by an intelligent system which provides explanations so that the system can serve as a positive instance of how examples can be explained. Two systems which demonstrate the example explanation feature are SCENT (McCalla & Greer, 1993) and the explanation environment used by Recker and Pirolli (1992). In both systems, the student can get interactive explanations of problem solving examples on several levels of granularity. However, these explanations are not adapted to the student's knowledge. In the user modeling domain, several technologies for generating explanations based on user models are discussed (Paris, 1989). Example explanation takes into account user knowledge on various topics mentioned in the explanation. In our case, such a technology will take into account all three kinds of knowledge about the user discussed in the paper.

Summary

This paper suggests and comprehensively discusses the idea of more adaptive intelligent learning environments for programming. We sustain the approach where the central student model, a traditional component of ITS, is used as a user model to adapt various components of an educational environment. We argue, that various kinds of knowledge about the user (concept level, episodic level, and goal-plan level knowledge) are required to support a reasonable range of adaptation techniques and, what is essential, advanced adaptation techniques. A student model which contains all types of knowledge mentioned above is ELM, which is briefly reviewed in the paper. We use the example of ALFRED, the structure editor of ELM/PE, to demonstrate several adaptation techniques based on various kinds of knowledge about the student reflected in the student model. Some of these techniques are already implemented and even evaluated, others are under construction. Our current work however, is not limited to making ALFRED adaptive. We are trying to move to a comprehensively adaptive learning environment where all components can use the ELM student model for the purpose of adaptation. We consider this way as a the way to the future of ITS.

References

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum Associates

Anderson, J. R. & Reiser, B. J. (1985). The LISP tutor. *Byte, 10*(4), 159-175.

Brusilovsky, P. (1992). Intelligent tutor, environment, and manual for introductory programming. *Educational and Training Technology International, 29*, 26-34.

Brusilovsky, P. (1993). Student as user: Towards an adaptive interface for an intelligent learning environment. In P. Brna, S. Ohlsson, & H. Pain (Eds.), *Proceedings of AI-ED 93* (pp. 386-393). Charlottesville, VA: AACE.

Corbett, A. T. & Anderson, J. R. (1992). Knowledge tracing in the ACT programming tutor. *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society* (pp. 623-627). Hillsdale, NJ: Cognitive Science Society.

Dieterich, H., Malinowski, U., Kühme, T., & Schneider-Hufschmidt, M. (1993). State of the art in adaptive user interfaces. In M. Schneider-Hufschmidt, T. Kühme, & U. Malinowski (Eds.), *Adaptive user interfaces* (pp. 13-48). Amsterdam: North-Holland.

Eisenstadt, M., Price, B., A., & Domingue, J. (1993). Redressing ITS fallacies via software visualization. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), *Cognitive models and intelligent environments for learning programming* (pp. 220-234). Berlin: Springer.

Fischer, G. (1988). Enhancing incremental learning process with knowledge-based systems. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligent tutoring systems* (pp. 138-163). New York: Springer-Verlag.

Herzog, C. (1992). From elementary knowledge schemes towards heuristic expertise—Designing an ITS in the field of parallel computing. In C. Frasson, G. Gauthier, & G. I. McCalla (Eds.), *Intelligent tutoring systems* (pp. 183-190). Berlin: Springer-Verlag.

Johnson, W. L. (1986). *Intention-based diagnosis of novice programming errors*. London: Pitman.

Kühme, T. (1993). User-Centered Approach to Adaptive User Interfaces. *Knowledge-Based Systems, 6*, 239-248.

McCalla, G. I. & Greer, J. E. (1993). Two and one-half approaches to helping novices learn recursion. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), *Cognitive models and intelligent environments for learning programming* (pp. 185-197). Berlin: Springer.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: a unifying view. *Machine Learning, 1*, 47-80.

Möbus, C., Thole, H.-J., & Schröder, O. (1993). Interactive support of planning in a functional, visual programming language. In P. Brna, S. Ohlsson, & H. Pain (Eds.), *Proceedings of AI-ED 93 - World Conference on Artificial Intelligence in Education* (pp. 362-359). Charlottesville, VA: AACE.

Paris, C. (1989). The use of explicit user models in a generation system for tailoring answers to the user's level of expertise. In A. Kobsa & W. Wahlster (Eds.), *User Models in Dialog Systems*. Berlin: Springer-Verlag.

Ramadhan, H. & du Boulay, B. (1993). Programming environments for novices. In E. Lemut, B. du Boulay, & G. Dettori (Ed.), *Cognitive models and intelligent environments for learning programming* (pp. 125-134). Berlin: Springer-Verlag.

Recker, M. M. & Pirolli, P. (1992). Student strategies for learning programming from a computational environment. In C. Frasson, G. Gauthier, & G. I. McCalla (Eds.), *Intelligent tutoring systems* (pp. 382-394). Berlin: Springer-Verlag.

Reiser, B. J., Kimberg, D. Y., Lovett, M. C., & Ranney, M. (1992). Knowledge representation and explanation in GIL, an intelligent tutor for programming. In J. H. Larkin & R. W. Chabay (Eds.), *Computer-assisted instruction and intelligent tutoring systems: Shared goals and complimentary approaches* (pp. 111-149). Hillsdale, NJ: Erlbaum.

Weber, G. (1994a). Examples and remindings in a case-based help system. In M. T. Keane, J. P. Haton, & M. Manago (Eds.), *Proceedings of the Second European Workshop on Case-based Reasoning (EWCBR-94)* (pp. 125-133). Paris: AcknoSoft Press.

Weber, G. (1994b). *Fallbasiertes Lernen und Analogien: Unterstützung von Problemlöse- und Lernprozessen in einem adaptiven Lernsystem*. Weinheim: Psychologie Verlags Union.

Weber, G. & Möllenberg, A. (1995). ELM programming environment: A tutoring system for LISP beginners. In K. F. Wender, F. Schmalhofer, & H.-D. Böcker (Eds.), *Cognition and computer programming* (pp. 373-408). Norwood, NJ: Ablex Publishing Corporation.