ARTIFICIAL INTELLIGENCE IN EDUCATION, 1993

edited by
Paul Brna
Stellan Ohisson
Helen Pain

Proceedings of AI ED 93
World Conference on
Artificial Intelligence in Education
Edinburgh, Scotland; 23-27 August 1993

AACE ASSOCIATION FOR THE ADVANCEMENT OF COMPUTING IN EDUCATION

Student as user: Towards an adaptive interface for an intelligent learning environment

P. BRUSILOVSKY
International Centre for Scientific and Technical Information
Kuusinen An 21b, Moscow 125252, Russia
E-Mail: plb@plb.icsti.su

Abstract: Most of intelligent learning environments provide the same interface for a novice and an experienced student, while the student's knowledge of the subject strongly differs in the beginning and in the end of the course. In this paper we discuss the problem of building an intelligent learning environment for programming, where the environment component can adapt its interface and performance to the student's knowledge level. The main idea suggested is to consider the student as the user of the environment and to use the student model as the user model for the purpose of adaptation.

Introduction

An intelligent learning environment is a relatively new kind of intelligent educational system. Traditional ITS structure comprises four modules (Wenger, 1987): the expert module, the pedagogical module, the student module and the interface. An intelligent learning environment (ILE) includes one more module, the environment module (Burton, 1988). "The term environment is used to refer to that part of the system specifying or supporting the activities that the student does and the methods available to the student to do those activities. That is, the environment defines the kinds of problems the student is to solve and the tools available for solving them." (Burton, 1988, p.109).

ILSs are able to support learning by being told, learning by doing and exploratory learning, thus combining the features of traditional ITSs and learning environments. While learning in an ILE a student spends most part of the time working with the environment. Most of the environments provide the same interface for a novice and an experienced student, while the student's knowledge of the subject strongly differs in the beginning and in the end of the course. In general, the interface of the environment is oriented to an experienced student. As a result, the interface appeares to be too complex for a novice, who does not have enough feedback and often bumps into unknown subject concepts. The problem is: how can the environments be constructed so that they have 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).

A good way to make an environment adaptive is to provide it with an intelligent self-adaptive interface (Benyon & Murray, 1988). There is a range of user characteristics that can be taken into account by an adaptive system to tailor the interface to the given user. The key characteristic of the student for an ILE is the student's knowledge level. Frye, Littman and Soloway (1988, p.453) stressed: "A further requirement for an interface to a piece of educational software is that it must be sensitive to the student's general knowledge and development level."

An adaptive interface can seriously improve the performance of ILEs that have a complex powerful environment component. Smithtown (Shute & Glaser, 1990), Sherlock (Lesgold et. al, 1990), and Quest (White & Frederiksen, 1990) provide good examples of modern powerful ILEs. A special kind of ILE with a complex and poverful environment component is an ILE for programming (we also call it Intelligent Programming Environments, IPEs). An IPE contains both, an intelligent coach or tutor which supports

student learning, and a programming environment which supports student programming activity. A number of IPEs have been described: APT (Corbett & Anderson, 1992), Bridge (Bonar & Cunningham, 1988), SCENT-3 (McCalla, Greer & SCENT Research Team, 1990), GIL (Reiser et al., 1992), ELM-PE (Weber & Moellenberg, in press), and Soda (Hohmann, Guzdial & Soloway, 1992). The following tools have been suggested by the IPE mentioned above to support a novice programmer: a plan-based program design tool, a language oriented structure editor, a visual program editor, tools for example-based programming, a compiler or interpreter with enhanced diagnosing possibilities, program visualizers (data visualizers and execution visualizers), on-line help about language features, on-line help about the environment itself. We think that some of the listed tools can improve the performance of the system by taking into account the student knowledge and adapting to it.

In this paper we investigate the problem of creating an adaptive interface for IPEs. In the next section we analyse some related fields that can serve as sources of ideas for adaptive IPE interface design. Then we suggest and discuss an approach to creating adaptive interfaces for IPEs. The main idea is to consider the student as the user of the environment and to use the student model as the user model for the purpose of adaptation. We describe ITEM/IP systems to illustrate the suggested approach and to demonstrate some ways of building adaptive components of ILEs on the basis of the student model. In the closing section we discuss some topics related to the main theme of this paper. In particular, we suggest a general student model-centered open architecture for building ILEs.

Related fields

The first attempt to build a programming environment which can take into account the state of the student knowledge is related to what we call the incremental approach to teaching programming languages (Brusilovsky, 1992c). In the incremental approach, a programming language is represented as a progressive sequence of language subsets. Each subset introduces new programming language constructs while retaining all the constructs of preceding subsets. A programming language is learned by examining progressively larger subsets of the language. The environment can take into account the current subset in order to improve error handling, error message generation, and program visualization. For example, the environment can provide more feedback and more information for the new features that are introduced in the current language subset, while never mentioning any concept from the following subsets. The incremental approach is an easy but powerful way to adapt a learning environment to the current state of the student's knowledge. Gerhard Fisher's (1988) investigations of the paradigm of "increasingly complex microworlds" and the work of White and Frederiksen (1990) on causal model progressions proved that the incremental approach can be applied effectively in a number of domains. However a problem arises when applying this approach to the domain of teaching a programming language (Brusilovsky, 1992c). The problem is that the incremental approach implies a pre-defined order of teaching the language concepts. On the other hand, there are a number of possible teaching orders for any particular programming language. A programming environment based on a pre-defined sequence of language subsets will be useful for those teachers who agree with the sequence. Such environments are not really adaptive. They provide "adaptation" to the student's knowledge by requiring the teacher and the student to adapt the order of learning to the pre-defined one, while really adaptive systems should adapt their work to any teaching order and any state of the student's knowledge.

The best experience in creating really adaptive systems was accumulated in the domain of intelligent tutoring systems (ITS) (Wenger, 1987). A characteristic shared by many ITSs is that they infer a model of the student's current understanding of the subject matter and use this individualized model to adapt the instruction to the student's needs (VanLehn, 1988). The student model performs a number of functions in various ITSs (Self, 1987; VanLehn, 1988). This provided good experience in using different forms of student models for adaptive support of teaching and learning. For example, the overlay student model appeared to be a good basis for adaptive curriculum sequencing and task sequencing (Brusilovsky, 1992x). Less investigated by ITS projects is adaptive support of doing. The oldest examples are the MALT (Koffman & Blont, 1975) and GCAI (Koffman & Perry, 1976) systems. These systems support solving a problem and entering the solution step-by-step in a dialogue mode. The overlay student model is used to vary the level of detail of the dialogue. For example, a poorly learned step of the solution is divided by the system into substeps, while a completely learned step is just passed over (the system

performs such a step itself). The goal of ITS is learning and the communication of knowledge (Wenger, 1987). The field of ITS suggests many good ideas concerning using student models for the adaptive support of learning and some ideas concerning adaptive support for solving a given problem for which the system knows the solution. To find more ideas concerning adaptive support of doing we have to learn some classes of systems that support doing.

Intelligent help systems (IHS) is a relatively new field, which has a deep roots in ITS research (Breuker, 1990). IHSs aim to support a user working with some computer system (for example, electronic mail system). IHSs provide the user with passive assistance (answering the student's questions) and active assistance (revealing wrong and suboptimal behavior and incrementally extending the student's knowledge). IHSs use overlay user models to tailor the answers and the explanations to each individual user's knowledge level. For example, in the explanations only concepts which the user actually knows are mentioned by SINIX Consultant (Nessen, 1989). Another example, EUROHELP (Breuker, 1990) and GENIE (Wolz, McKeown & Kaiser,1990) select either introducing or reminding strategies to provide explanations for a new or known concept respectively. The goal of IHSs is not to make the supported system adaptive, but instead to augment it with consulting behavior that makes its capabilities accessible to casual users. IHSs provide many good ideas for creating the adaptive intelligent help component of a learning environment, but no ideas about how to adapt other parts of the environment to the user.

The most relevant field that provides useful ideas for creating an adaptive learning environment is adaptive user interfaces (Benyon & Murray, 1988; Cooper, 1988). This is a relatively new (compared to ITSs) field, which studies interfaces that adapt themselves to suit the characteristics of the user. A key part of an adaptive interface is the user model which represents those characteristics of the user that are important for the purpose of adaptation. The model of user knowledge and the beliefs about the domain (the application system) is an important part of the general user model. A number of papers suggest useful techniques and ideas how to use the overlay model of user knowledge for the purpose of adaptation. For example, Sleeman (1985) describes how to adapt the explanation to the user's knowledge level by hiding concepts unknown to the user. Benyon and Murray (1988) suggest the adaptation of command-driven and script-driven systems by preventing the novice user from using certain commands and parameters, and by enabling and disabling subnetworks respectively. Cooper (1988) describes how to adapt the amount of feedback and guidance information in a dialogue by selecting one of multiple sub-dialogues for each of the subtasks on the basis of the user model.

As we can see from the analysis, ITSs, IHSs and adaptive interfaces use a student or user model for generally the same purpose of adaptation. This similarity results in about the same kind of student or user model in these systems. In general, it is the overlay model based on the domain model, where the domain is either the subject to be taught or the application system. For each element of the domain knowledge the student (user) model stores some data about the student's (user's) competence and previous experience with this element. For the purpose of bug diagnosis and explanation the overlay model can be augmented with the bug catalogue.

At the same time, the overall goals of these three kinds of systems are different: ITSs support teaching, while IHSs and adaptive interfaces support doing. As a result, the methods of support of the student and the user model are different. ITSs can use direct methods (tests and exercises) to keep the student model updated, while IHSs and adaptive interfaces have to rely on indirect methods, deriving the user level of competence from watching the user work with the application system.

Student as user: the unified student-user model

The analysis of the previous experience of creating adaptive systems does not provide us with examples of adaptive learning environments, but it does provide some ideas about how to create one. The simplest idea is to consider the learning environment as an application system and consider the student as the user of this application system. This approach is simple because the existing methods from the fields of adaptive interfaces and IHSs can be used to make the environment adaptive. According to this approach, an ILE is created from two rather independent components: the intelligent tutor which supports adaptive teaching on the basis of the student model and the environment which supports the student's programming activity, adapting the interface to the student's level of expertise with the help of the user model. Note that most existing IPEs have these components rather independent too.

To consider the student as the user is the first step towards an adaptive ILE. We suggest going further and applying the student model component of the programming tutor or coach as the user model component of the adaptive programming environment. This is really possible (because the student and user models for the same domain appeared to be similar), but requires additional effort to design a unified student-user model. This approach provides some advantages over the "simple" approach. First of all, an intelligent programming environment based on the unified model is a real integration (vs. a sum) of its components. The results of the student's work with one of components are reflected in the model and can be taken into account by other components to adapt their performance to the changes in the student's knowledge. Second, the quality of user modelling is improved, because the system can take the advantage of using both direct and indirect sources (vs. only indirect ones in adaptive interfaces) for keeping the unified model updated. Third, the overall amount of work to create a modelling component is decreased because one (vs. two separate) models have to be supported.

The use of the student model as the user model to support an adaptive interface of ILEs is a a really new but promising application of student models. It was not considered in the work on student modelling (Self, 1988; VanLehn, 1988), but it suggests a promising way to create more adaptive intelligent learning environments. To investigate the possibility and the methods of using a unified student-user model in ILEs for programming, we have designed two IPEs for introductory programming (ITEM/IP and ITEM/IP-II), which are described in the next section. We have tried to answer the questions about how a student model can be designed in such a way that it be used for the purpose of adaptation by both the tutor and the environment, and how the various components of the IPE can use this model to adapt their performance to the given user. One of the main objectives was to make as many modules of the systems as possible adaptive.

ITEM/IP: an example of adaptive programming environments

ITEM/IP stands for Intelligent Tutor, Environment and Manual for Introductory Programming. The systems ITEM/IP and ITEM/IP-II apply the mini-language approach to learning introductory programming (Brusilovsky, 1991). In the mini-language approach beginners learn what programming is, while learning how to use a simple mini-language to control a robot which acts in a microworld environment. ITEM/IP (Brusilovsky, 1992b) was designed in 1985 to support the introductory programming course based uponAhe educational mini-language TLringal (Brusilovsky, 1991) for the first year students of the Moscow University. ITEM/IP-II was designed more recently to support a part of "the computer literacy" course for 14-15 year-old students of Moscow schools. It uses another mini-language called Tortoise.

ITEM/IP systems consist of three main components: the programming laboratory, the tutoring component and the conceptual kernel. The programming laboratory includes a structural editor and visual interpreter for the mini-language. The interpreter produces an animated program execution, visually reflecting the flow of control and the result of any executed command on the "screen model" of the microworld. The tutoring component consists of the strategy module, the presentation module, the evaluation module and the repetition module. The strategy module compiles a list of all relevant teaching operations and selects the optimal teaching operation. The student can either agree with the suggested optimal operation, or select the next one himself from the list of relevant ones. The presentation module presents the teaching operation to the student according to the type of the operation. There are five kinds of teaching operations in both systems: presentation, example, test, problem solution example and problem to solve. The evaluation module analyses the results of the student's work with any teaching operation and updates the student model, thus keeping it actual. The repetition module provides the student with menu-based access to previously learned material: descriptions and examples. This module supports both on-line language help and example-based programming.

Most of the time when working with the system a student communicates with the programming laboratory to solve tasks or to investigate the mini-language constructs. At any time, a student can call the tutoring component for the next teaching operation, for descriptions and examples of previously learned concepts and constructs, or for examples of problem solutions.

The conceptual kernel includes the domain model, the student model and the base of teaching operations. The base contains three kinds of frames that are used to generate five kinds of teaching operations mentioned above. The domain model in ITEM/IP is a net with nodes corresponding to programming

concepts and mini-language constructs and with links reflecting several kinds of relations between nodes. The overlay student model reflects by a set of integer counters the extent to which the student has mastered the concept or the construct. In (Brusilovsky, 1992x) we described how the strategy module of the tutoring component uses the models to provide adaptive knowledge sequencing and task sequencing. In this paper we describe how some other modules use the student model to adapt the performance and the interface to the student.

Adaptive modules of ITEM/IP

The student model in ITEM/11? contains one integer counter for each domain model element. Four thresholds project the interval of each counter value into five subintervals defining five levels of knowledge for each element. The first and the last of the subintervals correspond to "unknown" and "learned" levels respectively. The other subintervals correspond to intermediate levels of knowledge. All the system modules, save the editor, take this information into account for the purpose of adaptation, but only the presentation module, the visual interpreter, and the repetion module tailor their interface to the level of student's knowledge. Note, that the strategy module uses all five levels. The other modules often distinguish four, three (unknown, new, learned) or even two levels.

The visual interpreter uses the student's current knowledge level to provide adaptive error handling and adaptive visualization. For each syntax or run-time error the interpreter has a choice of several diagnostic messages and one or more explanations at increasing levels of detail. Each message has a *condition,* a boolean expression which addresses the current knowledge levels of the constructs related to the error. When the interpreter reveals an error in the student program, all messages with true conditions are presented to the student. Thus, the more the construct is learned, the more concise an explanation message is presented. Still unknown constructs are never mentioned in the messages, which sometimes helps to understand an ambiguous error. For example the same error can provoke the message *undefined subroutine, if* the construct *of* subroutine is well-known, or the message unknown *statement,* if it is still unknown. If the concept is known but not studied completely, the first message is followed by a more detailed explanation.

The level of visualization granularity also depends on the student's knowledge. Most of the constructs have three or more levels of visualization granularity. As further constructs are learned, less visualization of detail is needed and less steps are spent to go through them. For example, when the construct of compound condition is introduced, it is visualized in details as a sequence *of* "steps" of interpretation. Each *of* elements of the compound conditions is visually checked in a separate step followed by visual effects, and the result (true or false) is also presented visually in a separate step. On the coarser level of visualization granularity, stepwise condition checking is substituted with single-step checking, then the visual effect of checking is switched off, and so on. If the compound condition is already studied completely (last level) it will not be visualized at all and no steps will be spent to check it - it just will be passed over by the flow of control pointer.

The repetition module uses the student model to compile three menus: the menu of known concepts and constructs, the menu of construct examples and the menu of task examples. An example is included in one *of* the latter menus *if* it was either previously completed by the student or it does not contain unlearned constructs and hence cannot be used as a test or problem to solve.

The presentation module uses the student model to generate a description *of* a concept or a construct when introducing or repeating it. A description is a presentation of textual information followed by a presentation of the relations which link the element presented with other domain knowledge elements. Note that construct examples are presented as a separate kind of teaching operation. The textual information which is stored for the given concept or construct can be divided into a sequence of text fragments. Each fragment has a condition which addresses the knowledge level of the given and related constructs. The information about these relations is not stored as text, but is generated using simple templates. While generating a description of the concept, the presentation module presents only the fragments with true condition and the relations with known knowledge elements. The more the concept or construct is learned, more concise (but more complete in the sense of relations) descriptions are presented. This method provides a smooth transition from learning to on-line help access.

Adaptive modules of ITEM/IP-II

The ITEM/IP-II project is not completed yet, however the working prototype is already in use in some Moscow schools. This prototype demonstrates some more examples of the student model-driven adaptive interface. First of all, ITEM/IP-II employs a more advanced student model. For each of the domain knowledge elements the student model supports a set of variables which reflects the student's previous experience with this element. Each module has its own projection (a simplified local view) of the student model. The projection is as rich as the module can use for its purposes. A set of rules is used to project the student model into each of the local views.

A new feature of ITEM/IP-II is a complete structure editor for the mini-language Tortoise. The editor enables the student to input the constructs using "hot keys" and the editor menu. Both these ways are tailored to the student knowledge level. The unknown constructs are not listed in the menu and can't be entered with hot keys. This limits the student's choices and prevents the student from bumping into unknown constructs by pressing the wrong key. The saved space in the menu is used to present the complete templates for the new constructs. If the construct is well-learned, only the main keyword is represented in the menu.

Another new feature is an extended adaptive program visualization. We implemented the same possibilities as in ITEM/IP, but added a special explanation window which is used to explain in words the result of each step of the program execution according to the semantics of the executed construct. For example, if the condition of the executed *while* construct is true, the flow of control pointer jumps into the loop and the following explanation is presented: "Since the condition of the while statement is true, we go to the first statement of the loop". As further constructs are learned, more concise explanations are presented and some steps "disappear" altogether. No explanations are presented for well-learned constructs. This explanatory visualization was added after classroom experience with ITEM/IP-II. It appeared that human assistants often work as such an "explanator" commenting in words on the "behavior" of recently introduced constructs in the given example or program.

Discussion

The modules of the ITEM/IP systems demonstrate some methods and ideas on how the information about the student's knowledge state reflected in the student model can be used to support not only adaptive teaching, but also adaptive interface in the ILE for programming. According to the implemented ideas, the amount of feedback and explanations decreases while the student goes from novice to experienced user. Informally, we can formulate the principles of adaptation in ITEM/IP systems as follows: (1) never mention (hide) unknown domain knowledge elements; (2) provide more feedback and explanations for new elements; (3) be concise in dealing with well-learned elements. The methods of adaptation used in ITEM/IP are relatively simple, as is the domain of introductory programming itself. Our goal was not to improve the known methods of adaptation, but to build a system where most of the modules can use the same student model to adapt their performance in various ways to the knowledge of the given user. On the further steps some simple methods of adaptation can be substituted by more complex effective technologies developed in the fields of intelligent interfaces and intelligent help systems.

An important problem not considered in this paper is the relevance of adaptation. The system can use very elaborate strategies to provide the student with the "optimal" next teaching operation, level of visualization, or help detail. The problem is whether the student agrees with the choice. The student could prefer another next operation, more (less) concise visualisation, or more (less) detailed help. To deal with this problem, we think, the adaptation should not be obtrusive, and the student should be provided with a choice: to agree with the decision of the system or to make his own choice. Our experience with task sequencing (Brusilovsky, 1992x) shows that novices tend to agree with the system choice, while experienced students often prefer their own choice. In ITEM/IP the student has a choice between adaptive and detailed visualization, between complete and adapted on-line help. To increase student freedom in ITEM/IP-II we plan to provide the student with the possibility to customize the level of adaptaion, thus enabling the student to build a projection of the student model "by hands".

Another problem that was not discussed is that the unified student-user model increases the basis of the student model support. The methods from all the three domains (ITS, IHS and intelligent interfaces)

can be applied to keep the single student model updated. In principle not only can each module of an ILE use the student model for the purpose of adaptation, but also each module can influence the student model, reflecting an experience that the student has demonstrated while working with this module.

These considerations lead us to the student model-centered view of the architecture of the ILE. The student model stands in the centre among the environment components (Figure 1). Each of the components (modules) can use and/or update the student model. Each component uses and updates its own view of the student model that we call *a projection,* representing the features of the student, which this module can use for the purpose of adaptation. Special sets of rules are used to project the central student model to each local view and to update the central model following the changes in the projection.

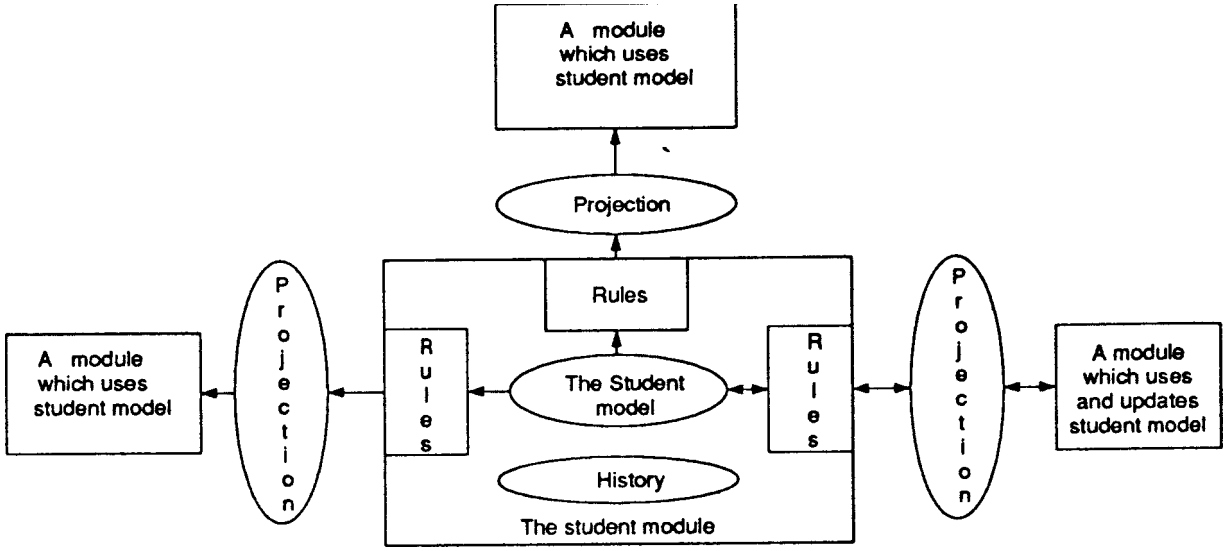

Figure 1: A student model centered architecture of ILE

We think that-this architecture is a good basis for creating an integrated ILE, as well as any other integrated adaptive system which consists of a set of more or less independent components. The use of projections and rules provides the architecture with good degree of flexibility and makes it open. *A* new component can be easily integrated into the environment by designing a set of rules that connect the central model with the local view of the given component. To investigate the possibilities of the suggested architecture we have used it as a basis for the ITEM/IP-II system.

In this paper we discuss the problem of building an intelligent learning environment for introductory programming, where the environment component can adapt its performance to the student's knowledge level. However, we think that the suggested ideas can be used not only in the field of programming but for creating ILEs in any domain.

References

Benyon D.R. k Murray D.M. (1988) Experience with adaptive interfaces. *The Computer Journal, 31(5), 465-473.*

Bonar J. k Cunningham R. (1988) Bridge: an intelligent tutor for thinking about programming. In Self J. (ed.) *Artificial intelligence and human learning. Intelligent computer-aided instruction.* Chapman and Hall. London.

Breuker J. (ed.) (1990) EUROHELP: Developing intelligent help systems. *Final Report on the P280 ESPRIT Project EUROHELP.* EC.

Brusilovsky P.L. (1991) Turingal - the language for teaching the principles of programming. In *Proceedings of the Third European Logo Conference.A.S.I.* Parma.

Brusilovsky P.L. (1992a) A framework for intelligent knowledge sequencing and task sequencing. In Intelligent Tutoring Systems. Proceedings of the Second International Conference, ITS'92. Springer-Verlag. Berlin.

Brusilovsky P.L. (1992b) Intelligent Tutor, Environment and Manual for Introductory Programming. *Educational and* Training Technology International, 29(1), 26-34.

Brusilovsky P.L. (1992c) Student models and flexible programming course sequencing. In Supplementary proceedings *of* the ICCAL'92, 4-th International Conference on Computers and Learning. Wolfville.

Burton R.R. (1988) The environment module of intelligent tutoring systems. In Polson M.C. & Richardson J.J. (eds.) Foundations of intelligent tutoring systems. Lawrence Erlbaum. Hillsdale.

Cooper M. (1988) Interfaces that adapt to user. In Self J. (ed.) Artificial intelligence and human learning. Intelligent computer-aided instruction. Chapman and Hall. London.

Corbett A.T. & Anderson J.R. (1992) Student modeling and mastery learning in a computer-based programming tutor. In Intelligent Tutoring Systems. Proceedings *of* the Second International Conference, ITS'92. Springer-Verlag. Berlin.

Fischer G. (1988) Enhancing incremental learning process with knowledge-based systems. In Mandl H. & Lesgold A. (eds.) Learning issues for intelligent tutoring systems. Springer-Verlag. New York.

Frye D., Littman D. & Soloway E. (1988) The next wave of problems in ITS: Confronting the "User Issues" of interface design and system evaluation. In Psotka J., Massey D.L. & Mutter S.A. (eds.) Intelligent tutoring systems: lessons learned. Lawrence Erlbaum. Hillsdale.

Hohmann L., Guzdial M. & Soloway E. (1992) SODA: a computer-aided design environment for the doing and learning of software design. In Computer Assisted Learning. Proceedings *of* the .4th International Conference, ICCAL'92. Springer-Verlag. Berlin.

Koffman E.B. & Blount S.E. (1975) Artificial intelligence and automatic programming in CAI. Artificial Intelligence , 6, 215-234.

Koffman E.B. & Perry J.M. (1976) A model for generative CAI and concept selection. International Journal on the *Man-Machine* Studies , 8, 397-410.

Lesgold A. et al. (1990) A coached practice environment for electronics troubleshooting. In Larkin J., Chabay R. & Scheftic C. (eds.) Computer assisted instruction and intelligent tutoring systems: *Establishing* communication and collaboration. Lawrence Erlbaum. Hillsdale.

McCalla G.I., Greer J.E. & Scent Research Team (1990) SCENT-3: An architecture for intelligent advising in problem-solving domains. In Frasson C., Gauthier G. & McCalla G.I. (eds.) Intelligent Tutoring Systems: At the crossrvads *of* artificial intelligence and education. Ablex Publishing. Norwood.

Nessen E. (1989) SCUM: user modelling in SINIX consultant. Applied Artificial Intelligence , 3, 33-44.

Reiser B. et'al. (1992) Knowledge representation and explanation in GIL, an intelligent tutor for programming. In Larkin J.H. & Chabay R.W. (eds.) Computer-assisted instruction and intelligent tutoring systems: *Shared* goals and complimentary approaches. Lawrence Erlbaum. Hillsdale.

Self J. (1987) Student Models: What Use are they? In Ercoli P. & Lewis R. (eds.) Artificial Intelligence tools in education. Proceedings *of* the The IFIP TC9 working conference on AI tools in *education.*

Shute V.J. & Glaser R. (1990) A large-scale evaluation of an intelligent discovery world: Smithtown. Interactive Learning Environments , 1, 51-77.

Sleeman D.H. (1985) UMFE: a user modeling front end system. International Journal on the *Man-Machine* Studies , 23, 71-88.

VanLehn K. (1988) Student models. In Polson M.C. & Richardson J.J. (eds.) *Foundations of* intelligent tutoring systems. Lawrence Erlbaum. Hillsdale.

Weber G. & Moellenberg A. (in press) ELM-Programming-Environment: A Tutoring System for LISP Beginners. In Wender K.F., Schmalhofer F. & Boecker H.D. (eds.) Cognition and computer programming. Ablex. Norwood.

Wenger E. (1987) Artificial intelligence and tutoring systems. Computational approaches to the communication of knowledge. Morgan Kaufmann. Los Altos.

White B.Y. & Frederiksen J.R. (1990) Causal model progressions as a foundation for intelligent learning environments. Artificial Intelligence , 42(1), 99-157.

Wolz U., McKeown K.R. & Kaiser G.E. (1989) Automated tutoring in interactive environments: A task-centered approach. *Machine Mediated* Learning , 3(1), 53-79.