Modeling the learner is a central aspect of intelligent tutoring systems and knowledge-based help systems that support learners in complex problem-solving domains. In this article, the episodic learner model ELM is introduced as a hybrid system that analyses novices' solutions to programming tasks based on both rule-based and case-based reasoning. ELM behaves like a human tutor. Initially, ELM is able to analyze problem solutions based only on its domain knowledge. With increasing knowledge about a particular learner captured in a dynamic episodic case base, it adapts to the learner's individual problem-solving behavior. Two simulation studies were performed to validate the system. The first study shows that the system can learn which rules are applied successfully to diagnose code produced by programmers and that using this information reduces the computational effort of diagnoses. Using information from the episodic learner model additionally speeds up the diagnostic process. The second study shows that ELM is able to predict individual solutions. Finally, correspondences and differences to related systems are discussed.

INTRODUCTION

Modeling the user in a so-called learner model is central to most existing intelligent tutoring or knowledge-based help systems. From the point of view of cognitive science, cognitive models that combine ideas from cognitive psychology with methods from artificial intelligence are most interesting. Despite the ongoing debate whether or not cognitive modeling is needed in ITSs (Newman, 1989; Swartout, 1985; Weber, Bögelsack, & Wender, 1993), it is accepted that cognitive modeling can be useful in complex problem-solving domains such as programming (Sandberg & Barnard, 1992).
In this article, we introduce the episodic learner model (ELM) that stores knowledge about one particular learner in terms of episodes. It supports the cognitive diagnosis of novices' solutions to LISP problems programmed in the knowledge-based help system ELM-PE (Weber & Mollenberg, 1995). The programming environment ELM-PE is aimed at helping novices to learn the programming language LISP.

First, we outline the idea of modeling the learner in ELM. We introduce episodic learner modeling in the scope of intelligent tutoring and knowledge-based help systems. Relations to explanation-based learning and to case-based learning help to clarify the principles ELM is based on. Second, we describe ELM in more detail. The architecture of the system, the diagnostic process in ELM, the storage and generalization of cases in ELM, and the use of information from the learner model are outlined and illustrated with examples. Third, we report on two simulation studies that try to validate the ELM model using data obtained from learners working with the programming environment ELM-PE. In the first study, simulations are run with different versions of the ELM system. Triggering the diagnostic process by using cases from the individual episodic case base is contrasted to selecting rules according to how often they were used successfully in previous analyses during a learning phase. In the second study, individual predictions of solutions to programming tasks with and without considering information from the episodic learner model are compared to solutions observed from learners during programming. Fourth, we discuss relations of ELM to other, already existing case-based reasoning and tutoring systems. In the last section, general conclusions are drawn about episodic learner modeling and experiences with using ELM in an intelligent learning environment are discussed.

Intelligent Tutoring and Human Cognition

Building an intelligent learning environment requires to integrate a model of human cognition into the knowledge-based program. Anderson (1984; Anderson, Boyle, Corbett, & Lewis, 1990) already stated that intelligent tutoring systems are suited to investigate human cognition in a natural problem-solving situation and to test out cognitive models. Therefore, it was not only our goal to write a program that is able to detect errors in problem solving (especially bugs in computer programs) but also to implement and to test a model of human cognition that describes human problem solving and the acquisition of procedural skills.

Our ideas of building such a cognitive model originate from Levine's investigations in concept learning (Levine, 1966). In his experiments, he could show that subjects follow the hypothesis they created to discriminate concepts. In the case of a contradiction to a hypothesis subjects tried to keep as many aspects of the old hypothesis as possible and changed only those parts.
of the hypothesis that were necessary to remove just this conflict. In the context of problem solving in learning programming, Weber and Bögelsack (1995) observed comparable learning mechanisms. When learning to code recursive programs in LISP, subjects had to solve a sequence of programming tasks that were preceded by two different examples relevant to these tasks. The first task could be solved in three different ways being directly related to one of the examples. In their attempts to solve the problem, subjects started with one of the three solution types and kept it until they came up with a final solution that was correct in most cases. Only 3 out of 26 subjects switched between different types of solution attempts. The second task could be solved in different ways, too, with each of them being related directly to one of the three solution types from the first task. All subjects tried to solve the second task in direct analogy to the solution to the first task, that is, they only changed few aspects of the old solution to adapt it to the new task. Only one subject switched to another solution type when her first attempt to solve the task according to the previous solution type failed. In the students' attempts to solve the programming tasks according to similar examples and previous solutions, typical errors occurred that, in many cases, could be traced back to taking over too many superficial aspects of the old solution. Weber and Bögelsack could show that ELM successfully simulated the subjects' problem-solving behavior.

Results from these studies show that subjects try to solve problems following previous problem solutions or examples. This is an economical procedure because it allows to keep as many aspects from the previous solution as possible and to change only the few things they believed to be necessary to solve the current problem. Case-based reasoning is assumed to be a method that is suited to model such a type of similarity-based problem solving and learning (Kolodner, 1993; Riesbeck & Schank, 1989). The episodic learner model ELM described in this article is based on such a case-based learning method.

**Intelligent Tutoring and Episodic Modeling**

Intelligent tutoring systems (ITS) and knowledge-based help systems are designed to support learners and users individually. Such a capability stems from two so-called *intelligent* properties. On the one side, an ITS is a knowledge-based system that is able to generate problem solutions automatically based on its domain knowledge without having to go back to inflexible, precompiled problem solutions (Self, 1974). This allows the system to analyze complex and uncommon problem solutions, and to identify and explain errors. Also, representing the learners' knowledge acquisition processes and their current knowledge in so-called *learner models* allows the system to adapt to the learners' needs. Such learner models depend on representing the learners' knowledge about the respective domain and on
describing their ability to solve problems in this domain. In existing tutoring systems, these intelligent properties (sometimes called qualitative models, see Clancey, 1986) involve, to varying degrees, automatic, cognitive diagnoses of activities and problem solutions offered by the learner. Results from these diagnoses are used for supporting learners and for giving advice in a communication process (Wenger, 1987).

Learner models based on overlays (Carr & Goldstein, 1977) identify the learner's performance as a subset of an expert's capabilities. They are adapted in their view of explaining solutions or errors from a learner to the point of view of the expert who planned or programmed the system (Ohlsson, 1986). But even models using bug libraries (e.g., the PROUST system, Johnson & Soloway, 1987), or generative, runnable models (e.g., the program space approach, Ohlsson, 1986, and the model-tracing method, Anderson, Conrad, & Corbett, 1989) are limited in their ability to take into account the learners' intentions or their personal problem-solving style. Individualizing helping actions and tutorial activities will effectively depend on two issues: (a) individual information about how a particular learner solved tasks should be kept for a long while, and (b) this knowledge must be used in subsequent diagnoses and tutorial decisions. How much and how long information has to be stored is an open question up to now.

The diagnostic process combines information from different components of the knowledge-based system and is implemented in many systems as a separate component, the diagnostic component. In the diagnostic process, the problem solutions offered by the learner or the actions planned by the user are analyzed with respect to the system's domain knowledge on the one hand, and the system's knowledge about the learner captured in the learner model on the other hand. Results from the diagnostic process can be used to explain errors to the learner, to give hints about possible plans and solutions, and to support the tutorial component's selection of appropriate exercises and questions. Additionally, the diagnostic result is used to update the individual learner model.

In knowledge-based help systems and in intelligent tutoring systems, a cognitive diagnosis aims at two different targets. On the one hand, based on its domain knowledge the system can decide whether the learners' solution is correct or not, and a possible error can be explained. This type of diagnosis depends on a runnable domain model (Self, 1974) and can already go far beyond both the simple diagnosis of syntactic errors in traditional man-machine systems and the offering of prestored answers and solutions to the learner (Nicolson, 1992). On the other hand, in the sense of the explanation component of an expert system, the cognitive diagnosis should be able to recognize which concepts and rules were used by the learner to solve the problem, and which errors and misconceptions led to an erroneous solution. In the end, information about the learners' knowledge (or lack thereof) enables the system to individualize its tutorial and helping activities.
Following these considerations, it is obvious that an efficient learner model can play a prominent role in individualizing helping actions and tutorial activities. In most intelligent tutoring systems, the tutorial component relies, in its tactical and strategic decisions, on information from an individual learner model. However, the more the diagnostic component is able to utilize individual information from a learner model while analyzing answers and problem solutions offered by the learner, the more it will be possible to offer explanations for correct or incorrect problem solutions regarding his or her individual problem-solving behavior.

In the episodic learner model ELM described in this article, individualization is accomplished by storing and generalizing information from sequences of learning episodes. From the resulting episodic traces, the system can demonstrate remindings to previous solutions from the user's learning history and to examples from the learning materials. Additionally, in the sense of a case-based learning system, episodic information can be used to trigger the diagnostic process and to reduce its computational effort. Thus, ELM is a hybrid system (Hammond & Hurwitz, 1988) combining ideas from case-based learning (Hammond, 1989; Kolodner, 1993; Riesbeck & Schank, 1989) with explanation-based learning (DeJong & Mooney, 1986; Mitchell, Keller, & Kedar-Cabelli, 1986). In the following subsections, we briefly describe rule-based diagnosis in ELM and outline the relation to both explanation-based learning and case-based learning.

**Rule-based Diagnosis in ELM**

Diagnosis of a solution to a programming task starts with a task description related to higher order concepts (general and LISP-specific programming concepts, plans, and schemata) in the knowledge base. Every concept comprises plan transformations and rules describing different ways to solve the goal given by the current plan. Without episodic information, rules are tested in the order in which they are stored in the concept frame. A rule is applied if its priority value is not lower than the overall priority of an existing competing interpretation in the same branch of the diagnosis or if it is the only rule that can be applied in that branch. Applying a rule results in comparing the plan description to the corresponding part of the student's code. In the plan description, further concepts may be addressed. This diagnostic cycle is called recursively until it matches a function name, a parameter, or a constant. The diagnosis results in a derivation tree built from all concepts and rules identified to explain the student's solution.

**Explanation-based Learning in ELM**

The interpretation and explanation of problem solutions is performed in ELM by a method that corresponds, in principle, to the explanation-based learning approach (DeJong, 1988; Mitchell et al., 1986). Explanation-based learning can be characterized as a type of learning by observation (DeJong
& Mooney, 1986). On the basis of common world knowledge and specific domain knowledge, the explanation-based learning system is able to acquire common knowledge from observing and explaining one or only a few examples. At first glance, one can take explanation-based learning as solving a complex problem that occurs repeatedly in the same or in similar ways. Furthermore, similar problems are identified to be solved with known complex problem-solving steps. That is, explanation-based learning means that in problem-solving situations that are similar to previous situations, operators are known that can be successfully applied to approach the solution. This is how the explanation-based learning method used in ELM works. Based on the domain knowledge of LISP programming, solutions and partial solutions to programming problems are explained. From these explanations, generalizations are built resulting in patterns for program code that can be matched during an analysis of a problem solution. These patterns are associated with rules that were applied successfully in previous problem-solving situations. If the episodic patterns match, the associated rules are preferred in the explanation process.

**Case-based Learning in ELM**

ELM stores information about a particular learner in terms of a collection of episodes. Such episodes can be called cases (Riesbeck & Schank, 1989). In the domain of learning to program, solutions to programming tasks represent episodes. These comprise examples the learners studied in the learning materials as well as their own solutions produced when working at exercises.

As ELM is a learning system, it not only diagnoses and explains the learners' programming activities, but it also builds up a case base, the learner model. In the beginning, analyzing the program code produced as a solution to a programming task depends solely on the system's knowledge about the programming task and on the domain knowledge about the programming language LISP. The domain knowledge comprises a canonical learner model including a bug library that describes bugs observed from students in previous programming courses. Gradually, ELM builds up the individual, dynamic learner model from information contained in explanation structures that result from diagnosing solutions to programming tasks. The explanation structures describe which knowledge structures—with respect to the system's domain knowledge—were used to generate the observed program.

Dividing the explanation structure into instances of concept frames results in a distributed storage of programming episodes. Episodes can be retrieved as a whole and, in addition, parts of previous solutions can be used directly to reconstruct explanation structures when analyzing new problems. The generalization hierarchies of episodic frames subsumed under a concept frame from the knowledge base reflect different degrees of organizational
similarity (Wolstencroft, 1989) for partial explanation structures. This type of organizational similarity between episodic frames can state a useful constraint on retrieving examples and remindings from the episodic learner model (Weber, 1995).

**DESCRIPTION OF ELM**

ELM is a type of learner model that stores knowledge about the learner in terms of a collection of episodes. To construct the learner model, the code produced by a learner is analyzed in terms of the domain knowledge on the one hand and a task description on the other hand (Figure 1). This cognitive diagnosis results in a derivation tree consisting of concepts and rules the learner might have used to solve the problem. These concepts and rules are instantiations of units from the knowledge base. The episodic learner model is built out of these instantiations and subsequent generalizations. What follows is a more detailed description of the knowledge representation and the diagnostic process.

**The Architecture of ELM**

The explanation step in ELM draws information from four sources: (a) the domain knowledge, (b) the description of the task the student is working on, (c) the individual episodic learner model, and (d) the LISP code produced by the student (Figure 1).

**Domain Knowledge**

The domain knowledge consists of concepts and rules that are stored hierarchically in a frame system. Rules are assigned to specific concepts. As a result, they can be applied in a specific context without having to search through a huge rule base. This hybrid representation is comparable to goal-oriented rules in the ACT models (Anderson, 1983, 1993) and to bundling rules under topics in CoPS (Tack, Wallach, Unz, Henss, & Engler, 1993).
Concepts

Concepts represent (a) knowledge of the programming language LISP (concrete LISP functions and procedures as well as superordinate semantic concepts), (b) knowledge of schemata for general algorithms and common problem-solving heuristics (e.g., specific recursion schemata), and (c) knowledge of meta-information which concern the diagnostic process itself. Slots of these concept frames contain information that can be used during the diagnostic process. As an example, some slots of the concept NIL-TEST are shown in Table 1. Two of these slots have to be discussed here. The Specializations slot contains a list of specializations, pointing to directly subordinate concepts and episodic instances. The Sorted-Rules slot contains a sorted list of rules that may be applied to the plan addressing this concept. When the knowledge base is built up, rule instances in this slot appear in random order. Rule instances can be sorted by their quality and, additionally, by their priority, frequency, recency, or activation depending on the type of model used in different simulation studies. In the current version of ELM-PE, rule instances are sorted by their priority.

Rules

In ELM, rules describe how programming concepts and schemata occurring in a problem-solving plan can be used by a programmer to produce program code. Rules can be of different quality. They describe (a) correct solutions ("good" rules), (b) solutions that are correct, but suboptimal to the problem at hand ("suboptimal" rules), and (c) buggy solutions related to typical errors observed in novices ("bug" rules). These bug rules constitute an error library in the sense of a canonical learner model (Rich, 1985) comparable to bug libraries in other tutoring systems (e.g., CMU-LISP Tutor, cf. Anderson, 1993, Anderson & Reiser, 1985; or PROUST, cf. Johnson, 1986).

Like concept frames, rule frames contain slots for their name, type, abstractions, and specializations. Additionally, specific rule slots describe quality, priority, activation, and the rule body of preconditions and consequences (see Table 2).
The quality is used to presort rules in the concept slot Sorted-Rules. Quality values are "good," "suboptimal," "bug," "new," and "unknown" in decreasing order. "Good," "suboptimal," and "bug" rules are internally sorted according to their priority, frequency, recency, or activation (see below). The order in which rules are tested is predetermined by their quality if no further information (e.g., rule frequency or episodic information) is available. It starts with the highest quality "good." The Default-Rule with quality "unknown" applies only if the corresponding part in the analyzed program code cannot be explained by any other rule. That means that the explanation, at least for this part of the derivation tree, failed. In case a default rule applied, a new rule is created with quality "new." This is described later in more detail. The slots for priority, frequency, recency, and activation define a finer grain size for ordering "good," "suboptimal," and "bug" rules. Priority values are defined in advance by the developer of the system while values for the other slots can be preset with values learned by simulation runs with data from other students and change with diagnoses. The Precondition and the Consequence slots describe the precondition and the consequences parts of the rule. These slots are described in more detail later on.

In addition to these slots, templates for help texts are stored in a separate rule slot. These templates are filled according to the context in which the rule applied when the system gives feedback to the learner. An example can be found in Weber and Möllenberg (1995).

**Task Description**

Task descriptions contain—among other things—information about the text of the task, examples of I/O-description, and one or more descriptions of algorithms to solve the task. Information from the task description is used by different modules of the programming environment ELM-PE.
TABLE 3  
Parts of the Task Description for the Recursive Problem "Simple-And"

<table>
<thead>
<tr>
<th>Name:</th>
<th>Simple-And</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text:</td>
<td>&quot;Define the function SIMPLE-AND. Its argument must be a list of truth values (Ts and NILs). SIMPLE-AND tests if the value of all list elements is T (or non-NIL). SIMPLE-AND then returns T, else NIL.&quot;</td>
</tr>
<tr>
<td>I/O.Descr:</td>
<td>(T T T T) --&gt; T, (T NIL T) --&gt; NIL, (NIL NIL NIL) --&gt; NIL, () --&gt; T</td>
</tr>
<tr>
<td>Algorithm:</td>
<td>1</td>
</tr>
<tr>
<td>Ref. Sol.:</td>
<td>(defun simple-and (li)</td>
</tr>
<tr>
<td></td>
<td>(cond ((null li) t)</td>
</tr>
<tr>
<td></td>
<td>((not (car li)) nil)</td>
</tr>
<tr>
<td></td>
<td>(t (simple-and (cdr li)))))</td>
</tr>
<tr>
<td>Type:</td>
<td>DEFINE-PROCEDURE</td>
</tr>
<tr>
<td>Params:</td>
<td>(?LIST)</td>
</tr>
<tr>
<td>Body:</td>
<td>CDR-END-RECURSION</td>
</tr>
<tr>
<td>Rec.-Parameter:</td>
<td>?LIST</td>
</tr>
<tr>
<td>Case 1:</td>
<td>Test: (NULL-TEST (PARAMETER ?LIST))</td>
</tr>
<tr>
<td></td>
<td>Consequence: (TRUTH-VALUE T)</td>
</tr>
<tr>
<td>Case 2:</td>
<td>Test: (NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST)))</td>
</tr>
<tr>
<td></td>
<td>Consequence: (TRUTH-VALUE NIL)</td>
</tr>
<tr>
<td>Case 3:</td>
<td>Test: (ELSE-TEST T)</td>
</tr>
<tr>
<td></td>
<td>Consequence: (CDR-REC-CLAUSE (PARAMETER ?LIST))</td>
</tr>
</tbody>
</table>

(Weber & Möllenberg, 1995). The slots Type, Params, and Body from the different algorithms (see Table 3) are relevant to the diagnostic component of ELM. The Type slot indicates whether the goal of the task is to program a LISP-expression or to program a function definition. The Params slot lists the names of the arguments of a function definition that are referenced in the body of the function. The Body slot contains a higher level description of a plan addressing LISP-concepts and higher programming concepts (e.g., schemata) from the knowledge base of ELM. In Table 3, parts of the task and plan description for a simple end-recursive function definition are shown. The highest plan in the Body slot addresses a cdr-end-recursion schema that can be seen as a special case of a universal part-rest-recursion schema (Vorberg & Goebel, 1991). Plans describing algorithms for solving programming tasks are hierarchically organized. They address concepts from the system's knowledge base with subplans or constants as their arguments. For instance, the plan for solving the problem "Simple-And" (Table 3) specifies the plan (NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST))) for the test in the second case of the case decision. This plan addresses the higher programming concept NIL-TEST from the knowledge base with the subplan (FIRST-ELEMENT (PARAMETER ?LIST)) as its argument.

**Learner Model**

The learner model consists of information about a particular learner. Concepts registered in the derivation tree from the diagnosis of program code
for a programming task are inserted into the knowledge base as instances of their respective concepts. An episodic instance contains information from the derivation tree about the plan that addressed this concept and about the rule applied to solve this plan. Thus, traces of programming episodes are distributed over the knowledge base in the sense of an overlay model (Carr & Goldstein, 1977). The complete mechanism of building up the learner model will be described in more detail below.

**Code for the Problem Solution**

The LISP code of the solution to a programming task should be coded in a LISP structure editor (Köhne & Weber, 1987; Weber, 1993). Thus, the program code is at least syntactically correct and allows the system to analyze incomplete solutions. In the case of incomplete but correct solutions, the system will give a hint as to how to complete the code.

**Diagnosis in ELM**

The automatic diagnosis operates similar to the problem-space approach (Ohlsson, 1986). Based on the domain knowledge, on the learner model, and on knowledge about the task, the system tries to generate the same code that the learner produced. In terms of the EBG method (Mitchell et al., 1986), the code produced by the learner plays the role of a training example. All plans, concepts, and rules used to generate the code automatically are collected in a derivation tree. This derivation tree explains, according to the explanation step in the EBG method, how the code could be produced by the learner from the point of view of a tutor. Suboptimal and bug rules identified during the diagnosis indicate the learners' misconceptions and errors. Thus, one can speak of a cognitive diagnosis in ELM.

Before starting the cognitive diagnosis, a fast pre-analysis tries to identify the algorithm used in the solution, the order of arguments in a function definition in relation to the order stated in the plan description, and the order of cases in case decisions (e.g., in recursive function definitions). The algorithm chosen by the learner to solve the problem is identified by heuristic rules. They refer to reference solutions from the task description (Table 3). In most cases, however, only one algorithm is given in the task description with a plan description that is flexible enough to cover most solutions observed from typical learners. The pre-analysis is partially done by a simple version of the TALUS debugger (Murray, 1988). It is described in more detail in Weber and Möllenberg (1995).

The cognitive diagnosis starts with the plan description for the algorithm determined during the pre-analysis. The plan description consists of plans addressing concepts from the knowledge base. The goal is to generate the code offered by the learner by applying rules and transformations that are stored with the concept addressed by the plan.
Applying Rules
In each concept frame, a list of rules is stated showing different ways to solve the plan that addressed this concept. Rules are sorted by their quality value classifying them into different quality categories. Rules are tested in the sequence in which they appear in the list of sorted rules. That is, starting with the highest quality, the system tries to follow the path shown by the rule. If all nested plans are solved with the same or even with a higher quality, no rules with lower quality are tested. However, all rules with the same priority are tested so that, in principle, multiple explanations can be computed within one single analysis. The preset order of rules given by their priority can be altered by their current frequency, recency, or activity and by episodic information as described in a subsequent section.

Before trying to apply a rule, its preconditions are tested. Many rules do not require more than the implicitly fulfilled precondition that the plan to be solved addressed the concept where this rule belongs to (e.g., Equal-NIL-Test-Rule, Table 2). More specific preconditions can ask for certain subplans as arguments or for a definite number of arguments in the current plan, for specific code templates, etc. If all preconditions are met, actions from the consequences part of the rule are executed. Actions may consist of solving an alternative plan (see Table 2) or of solving a sequence of subplans. The latter case can be easily explained using the Unary-Func-Rule, the common rule for applying a unary LISP-function (e.g., for the concept FIRST-ELEMENT). In this rule, two actions are executed sequentially. In the first action, a plan to test for the FIRST-ELEMENT-OPERATOR is stated. In the operator concept frames, rules are contained that match the name of the LISP function in the corresponding part of the code. For the FIRST-ELEMENT-OPERATOR, a rule exits that tests whether the function names first or car are coded. Other rules may look for a typical confusion of names or for other types of errors. In the second action, the subplan to code the argument of the plan that addressed the concept FIRST-ELEMENT has to be solved.

Following the algorithm, the diagnosis of the code works recursively. Starting from the plan description for the task, a concept from the knowledge base is addressed that contains rules and transformations to solve the plan by setting up a new plan or a sequence of subplans that in turn address concepts, and so forth. This recursive process terminates when a rule for a subplan successfully matches a function name, a function parameter, or an expected constant, or, if no rule matches, the default rule applies. The result of this analysis is a derivation tree (Weber, 1988) consisting of all concepts and rules used to automatically construct the code.

Creating New Rules
In a few cases, a very uncommon but correct solution to a programming task may be coded or an unforeseen error happens that cannot be explained
by an existing rule. Then the system is unable to give detailed feedback, at least for that part of the code to which no rule could be applied successfully. In such a situation, the Default-Rule with the quality "unknown" is included into the derivation tree, indicating that for the current plan no rule with a higher quality can be applied. This Default-Rule contains a template for a help text allowing the help system to give feedback to the programmer. It describes that at the corresponding place in the program code an error was detected, what the correct plan was to follow at that place, and, if requested, how the program could be coded correctly at that place. Such a feedback, however, does not explain the type of error and how such an error could have happened.

For each Default-Rule in a derivation tree, a new rule is created and inserted into the list of rules from the corresponding concept. It is applied in a similar situation when the same plan is solved with the same code (except for differences in variable names). Such a new rule does not give an explanation comparable to other rules. Rather, it indicates that the same situation happened before, and a help text mentions this fact. For the developers of the system, the appearance of such a new rule can be a hint to analyze the underlying problem and to add appropriate new rules to the system.

Transformations
In a concept frame, plan transformations may be stated representing semantically equivalent variations of plans that may address this particular concept. For example, transformations allow changes in the order of a sequence of cases in a case decision or a switch between logically equivalent plans. The mechanism of transforming plans is described in more detail in Weber (1994). Plans are only transformed if the original plan fails. If there are two or more plan transformations stored with a concept, those transformations are preferred that are directly indexed by the corresponding part of the program code. In contrast to rules, plan transformations do not have priorities. Transformations can be sorted according to their syntactical similarity to the program code. In case of information from the episodic learner model, however, those transformations can be preferred that were successfully applied in a previous episode in a similar situation.

Diagnosis in ELM: An Example
Building up the derivation tree during a diagnosis can be demonstrated using a more detailed example. In the LISP introductory course, the recursive programming problem Simple-And (Table 3) had to be solved as the first exercise in the first lesson on recursion. A typical solution to this problem observed from some students was the following code:

```lisp
(defun simple-and (li)
    (cond
        ((null li) t)
        ((equal (car li) nil) nil)
        (t (simple-and (cdr li)))))
```
The analysis of this solution starts with matching the head of the function definition containing the name for the LISP-procedure `defun`, the name of the function, and the parameter list. The body of the function definition is described by a plan to solve the CDR-END-RECURSION schema. To match this plan, in a first step the plans for the tests from each case (NULL-TEST (PARAMETER ?LIST)), (NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST))), and (ELSE-TEST T) are matched against the corresponding code fragments (null Ii), (equal (car Ii) nil), and t, respectively. This matching process can detect interchanges in the sequence of cases. In a second step, the consequences from each case are matched.

Starting with the first case of the case decision, the plan (NULL-TEST (PARAMETER ?LIST)) is matched against (null Ii). The concept NULL-TEST represents a test of whether its argument is an empty list. In the context of a list recursion, this means that the recursion list is exhausted. For a correct solution to this plan, the rule that is usually preferred and, therefore, has the highest frequency value will be considered first. This is the Unary-Function-Rule. In its specialization to the concept NULL-TEST, two subplans have to be solved. The first subplan (NULL-TEST-OPERATOR) checks whether the test on the empty list was coded with one of the LISP functions null or endp. The second subplan (PARAMETER ?LIST) checks whether the correct parameter from the parameter list of the function definition was coded.

In this analysis, the test in the second case of the case decision is most interesting. The plan for this test (NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST))) expects the first element of the list bound to the parameter ?LIST to have the truth-value NIL. In the solution, this test was coded correctly with the expression (equal (car Ii) nil). The derivation tree for this part of the diagnosis is shown in Figure 2.

The plan (NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST))) addresses the concept NIL-TEST. In the concept frame, several rules are listed describing different ways to solve the plan correctly, suboptimally, and erroneously. Names of the "good" and "suboptimal" rules are shown in Table 1. For this student it was the first time the concept NIL-TEST was encountered in a diagnosis. Therefore, no episodic information about using this concept existed. Hence, rules are tested in the order given in the Sorted-Rules slot in the concept NIL-TEST. First, the "good" Negation-NIL-Test-Rule was tried. The test in the precondition of this rule failed because the code (equal (car Ii) nil) does not start with not.

The next rule selected was the suboptimal Equal-NIL-Test-Rule (Table 2). It sets up a new plan (EQUALITY (FIRST-ELEMENT (PARAMETER ?LIST)) (TRUTH-VALUE NIL)) to be solved. This plan addresses the concept EQUALITY whose frame contains rules for coding binary commutative functions. The first rule in the sequence of Sorted-Rules, the Binary-Function-Rule, was applied. As a consequence, a sequence of subplans had to be evaluated. First, correct coding of an EQUAL-OPERATOR had to be tested,
subsequently, the plans (FIRST-ELEMENT (PARAMETER ?LIST)) and (TRUTH-VALUE NIL) had to be solved. All these plans and all subordinated plans could be explained directly by rules with the highest priority. Thus, no other rule from the concept NIL-TEST with a lower priority was tested. The resulting derivation tree is shown in Figure 2.

In the last clause of the case decision, the plan (ELSE-TEST T) was successfully matched against t. With that the analysis of tests in the case decision was completed and it remained to match the plans from the action part of the clauses with the corresponding code. The truth-values in the consequences part of the first two clauses as well as the recursive call in the consequence part of the last clause were correctly coded and could easily be checked.

The complete derivation tree is returned as the result of the cognitive diagnosis. The tree contains the complete information about all plans followed to solve the programming problem, about all concepts that were addressed by these plans, and about all rules that were applied to fulfill the plans. In other words, it is explained how the observed program code could have been generated during the problem solving process. This information is used by the system to update the learner model. A tutorial component can use this information to give feedback to the programmer. In ELM-PE, text patterns stored with concept and rule frames can be used to give feedback in cases of suboptimal or buggy solutions. A more detailed example of giving feedback in ELM-PE can be found in Weber (1994) and Weber and Möllenberg (1995).
TABLE 4
Some Slots of the Frame NIL-TEST.TS.4-1 (Before Generalization)

<table>
<thead>
<tr>
<th>Name:</th>
<th>NIL-TEST.TS.4-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>EPI-Instance</td>
</tr>
<tr>
<td>Episode-No:</td>
<td>4</td>
</tr>
<tr>
<td>Event-No:</td>
<td>1</td>
</tr>
<tr>
<td>Task</td>
<td>Simple-And</td>
</tr>
<tr>
<td>Context:</td>
<td>(2.-Case (NIL-TEST ...))</td>
</tr>
<tr>
<td>Concept:</td>
<td>NIL-TEST</td>
</tr>
<tr>
<td>Abstraction:</td>
<td>NIL-TEST</td>
</tr>
<tr>
<td>Transformation:</td>
<td>nil</td>
</tr>
<tr>
<td>Plan:</td>
<td>(NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST)))</td>
</tr>
<tr>
<td>Datum:</td>
<td>(equal (car li) nil)</td>
</tr>
<tr>
<td>Rule:</td>
<td>Equal-Nil-Test-Rule</td>
</tr>
<tr>
<td>Rule-Quality:</td>
<td>suboptimal</td>
</tr>
<tr>
<td>Overall-Quality:</td>
<td>suboptimal</td>
</tr>
<tr>
<td>Overall-Priority:</td>
<td>6</td>
</tr>
</tbody>
</table>

Storage and Generalization of Cases in ELM
In the second step of diagnosis in ELM, results from the cognitive diagnosis are stored and generalized in the episodic learner model. For each concept shown in the derivation tree, an episodic instance is created and subsumed under the concept from the knowledge base. In case of an already existing hierarchy of episodic and generalized frames under this concept, the episodic frame is inserted into the hierarchy at a level at which (a) the superordinate generalization frame is most specific to this episodic instance and (b) all neighbors on the same level differ in one or more aspects. For all frames on this level, commonalties in the sense of similarity-based generalization (Lebowitz, 1983) are collected, and, if possible, the most similar frames are generalized. If an instance is the first one subsumed under the concept frame of the knowledge base, this episodic frame will be generalized in the sense of the EBG method (Mitchell et al., 1986).

Inserting new episodic frames into the episodic learner model and the subsequent explanation-based generalization will be demonstrated using the concept NIL-TEST from the example of the analysis of the program code for the task "Simple-And." For the concept NIL-TEST noted in the derivation tree (Figure 2), the episodic instance NIL-TEST.TS.4-1 is created (see Table 4). In the instance frame, it is noted that it was the first event in the fourth episode. Additionally, it is noted that the concept NIL-TEST was addressed

1In ELM-PE it is assumed that the programmer has examined three examples of recursive function definitions from the course materials for this lesson in advance. Thus, when starting to work on this task, the solutions for the examples from this lesson were analyzed by the diagnostic component and the resulting derivation trees were integrated into the learner model.
in the context of the test in the second case of a case decision by the plan
(NIL-Test (First-Element (Parameter ?List))). The corresponding code
from the problem solution (equal (car li) nil) was explained by the "sub-optimal" Equal-NIL-Test-Rule.

Explanation-based Generalization
Whenever an episodic instance is subsumed directly under the concept
frame from the knowledge base, it will be generalized according to the EBG
generalization step. With this type of generalization, the part of the pro-
gram code solving the plan that addressed this concept is generalized with
respect to knowledge about LISP. It results in a code template describing a
class of code that can be matched by the same rule as noted in the episodic
instance. This will be explained using the episodic instance NIL-TEST.TS.4-1
(Table 4).

The episodic frame NIL-TEST.TS.4-1 is the first instance under the con-
cept NIL-TEST, so it will be generalized directly. Generalizing the code
(equal (car li) nil) results in a template (<Equal-Op> (<First-Elem-Op> <Variable>) nil). The function name equal is generalized to <Equal-Op>,
the class of all operators that are semantically equivalent to equal. The
pattern <First-Elem-Op> stands for all LISP operators that access the
first element of a list (e.g., car, first, and every self-defined function assessing
the first element of a list). The pattern <Variable> indicates that a
parameter from the function's parameter list should appear at this place.
All code fragments matching this template can be processed starting with
the rule Equal-NIL-Test-Rule. The code template is stored in a generaliza-
tion frame NIL-TEST.S.4-1.EBG (Table 5) that will be inserted into the
hierarchy between the episodic frame NIL-TEST.TS.4-1 and the concept
frame NIL-TEST (Figure 3). The Abstraction slot in the episodic frame
NIL-TEST.TS.4-1 (Table 4) is now linked to the new superordinated frame
NIL-TEST.S.4-1.EBG.
Figure 3. Part of the hierarchy of episodic instances and generalizations after inserting and generalizing the episodic frame NIL-TEST.TS.4-1.
Replacing code fragments by variables results in a very specific template (DeJong, 1988). This is typical of explanation-based generalization. Thus, further examples of code may not fit this pattern and a more general template has to be generated by similarity-based generalization.

**Similarity-based Generalization**

When a new episodic instance is inserted into an already existing frame hierarchy, similarity-based generalization can be applied. Episodic instances as well as generalization frames adjoined at the same level can be generalized. Similarity-based generalization can refer to information from the Context, Plan, Datum, and Rule slots. In principle, the similarity-based generalization mechanism implemented in ELM can result in specializations in the sense of a more specific generalization under an already existing generalization frame (described in Weber and Bögelsack, 1995) as well as in more general frames above more specific generalization frames (e.g., the generalization frame NULLTEST.G-3-1 in Figure 3). In the current version of ELM-PE, however, generalizations in the sense of specializations are not created because the number of frames in the episodic learner model would be inflated without improving or speeding up the diagnostic process.

**The Importance of Generalizations in ELM**

During diagnosis, generalized frames help to shorten the process of searching for rules that can be episodically preferred for the current plan. Instead of scanning all episodic frames, search can start from the concept frame. Subsequently, the search process goes down all paths to generalization frames similar to a discrimination net as long as plans and data match. It stops when a rule is found that can be applied. Code templates in generalization frames as well as code in episodic frames play the role of preconditions in selecting rules during the diagnostic process. In this way, the system learns which rules will apply in a known situation.

Generalization hierarchies built up from generalizing plans reflect structural dependencies and similarities of problem solutions for different programming problems. Together with semantic links between concepts and rules from the knowledge base, these hierarchies represent organizational similarities (Wolstencroft, 1989) that are used in the EBR method (Weber, 1991, 1993) to retrieve examples and remindings.

**Using Information from the Learner Model**

The case-based reasoning approach in ELM is aimed at three different goals. First, episodic information can be used to find and to show examples as well as remindings from the students’ own learning history. This important feature of ELM has been described elsewhere (Weber, 1991, 1994). Second, the episodic learner model contains much information that could be utilized
by a tutorial component of an ITS (a) to select exercises that are appropriate to the students' needs and (b) to adapt the giving of hints to the students' assumed knowledge level. As of yet, such an explicit tutorial component has not been included in ELM-PE. And third, diagnosing problem solutions should be improved and shortened by finding solution paths that are stored with solutions to similar previous cases and by reconstructing partial previous solutions according to *shortcuts* (Hammond, 1989; Strube & Janetzko, 1990). This aspect will be discussed in more detail now.

Two principally different situations can be distinguished when using episodic information during the diagnostic process. First, a partial solution of a task may be identical (except for variable names) to a partial solution from a previous episode in the same problem-solving context. In this case, the partial derivation tree from the previous episode can simply be reconstructed from the episodic frames stored in the episodic learner model. This mechanism of shortening the problem-solving process is comparable to creating and using chunks in SOAR (Laird, Rosenbloom, & Newell, 1985; Newell, 1990). In terms of case-based reasoning, this corresponds to reusing a partial previous solution with *null-adaptation* (Riesbeck & Schank, 1989).

Second, a plan and/or code may not match exactly the corresponding information from episodic frames. However, information from episodic frames indicates which rules were applied successfully in previous, similar situations. In this case, all rules from the frame of the concept addressed by the current plan are categorized into two different classes. The first class comprises all rules indexed by episodic and generalization frames whose plan and datum patterns match the current plan and the current datum. As described earlier, plan and datum patterns from episodic frames play the role of additional rule-preconditions that are met by the current situation. Rules from this group are preferred when trying to solve the current plan. In case-based reasoning systems, this adaptation technique is called *reinstatiation* (Riesbeck & Schank, 1989). Within this quality class, rules can be sorted according to the current values in their Frequency, Recency, or Activation slots, and, in case of ties, additionally by their priority value. The second class comprises all other rules that are not indexed episodically. Also, rules from this class can be sorted by numeric values as above.

The algorithm of categorizing and sorting rules described above considers two different types of information: symbolic information from episodic frames and numeric frequency, recency, or activation values from rule frames. Using numeric values is common to many learner models (e.g., aggregating the number of times a rule was applied successfully or not as in the CMU-LISP-Tutor, Anderson, 1993; Anderson et al., 1989; Corbett & Anderson, 1992). However, up to now storing and describing the circumstances under which a rule was applied successfully has not been implemented in systems using individual learner models. So the question arises as
to whether symbolic episodic information will have an advantage over simply considering accumulated numeric information. In the following, an example demonstrates how using episodic information influences the diagnostic process.

**Considering Episodic Information in the Diagnostic Process**

The mechanism of considering episodic information during the diagnostic process can best be explained using the example described earlier. After solving the problem "Simple-And" the student tried to solve the next exercise "Simple-Or." The predicate "Simple-Or" has to test whether its argument, a list, contains at least one truth-value "T." Instead of performing a NIL-TEST in the test of the second case of the case decision from the task "Simple-And," in this task the concept T-TEST applies, testing whether the first element of the recursion list has the truth-value "T." The following code is typical for many solutions observed from students that solved the problem "Simple-And" similar to the definition described above in our introductory courses (Weber and Bögelsack, 1995):

```lisp
(defun simple-or (li)
  (cond ((null li) nil)
        (((not (equal (car li) nil) ) t)
        (t (simple-or (cdr li) ) ) ) ) )
```

Most interestingly, in this solution the test on the truth-value "T" is solved by a negation of the test on the truth-value "NIL." Thus, the corresponding part of the code from the solution to the problem "Simple-And" could be reused. This is a typical example of how problems are solved in analogy to a previous problem with minimal changes to the code. Many students indeed produced errors by copying too many parts of the previous solution (e.g., copying the truth-values in the consequences of the first two cases) without considering that these parts had to be changed as well (see Weber & Bögelsack, 1995; for more empirical work on the issue of code re-use and analogy see Escott & McCalla, 1988).

The diagnosis of the code (not (equal (car li) nil) ) for the test in the second cond-clause starts with the plan (T-TEST (FIRST-ELEMENT (PARAMETER ?LIST))) from the plan description. In the concept frame T-TEST, four specific rules are listed that may be applied to solve plans addressing this concept. There are two "good" rules (*Simple-T-Test-Rule* and *Equal-T-Test-Rule*), one suboptimal rule (*Test-With-Double-Negation-Rule*) and one rule describing a typical error for this concept (*False-Equal-T-Test-With-NIL-Rule*). The concept T-TEST is addressed the first time in this diagnosis, so no episodic instances exist and rules are tested in the order given in the Sorted-Rules slot. The "good" rules are tried first. The *Simple-T-Test-Rule* tests whether the plan (FIRST-ELEMENT (PARAMETER ?LIST)) for the argument of the T-TEST was solved by coding directly the
first element of the recursion list. Because every expression in LISP represents a truth-value, this is the simplest way to test on the truth-value "T." The Equal-T-Test-Rule tests whether the argument of the T-TEST was directly compared to the truth-value "T" with the function equal (comparable to the Equal-NIL-Test-Rule described above). Neither rule can be applied successfully without assuming a coding error. Next, the suboptimal T-Test-With-Double-Negation-Rule is tried. This rule requires as a precondition that the corresponding part of the function definition starts with coding the logical function not. Because this is true, the rule sets up a new plan \((\text{NOT} \ (\text{NIL}-\text{TEST} \ (\text{FIRST}-\text{ELEMENT} \ (\text{PARAMETER} \ ?\text{LIST}))))\) to be solved. This new plan addresses the concept NOT with the subplan \((\text{NIL}-\text{TEST} \ (\text{FIRST}-\text{ELEMENT} \ (\text{PARAMETER} \ ?\text{LIST}))))\) as its argument. This plan can be solved directly with theUnary-Func-Rule (Figure 4), the first rule in the concept NOT that is used normally for coding a unary function. This rule sets up two new subplans, first a plan to code a negation operator (e.g.,
The negation operator is coded correctly. Therefore, the second subplan is tried next. For the concept NIL-TEST, an episodic instance as well as an EBG-frame exist from the previous diagnosis (see Figure 3). The generalization frame NIL-TEST.S-4-1.EBG (Table 5) contains a code template (\(<\text{Equal-Op}\>\ (<\text{First-Elem-Op}\>\ <\text{Variable}\>)\ nil) that matches to the code (equal (car li) nil). This indicates that the suboptimal \textit{Equal-NIL-Test-Rule} should be preferred to other rules, even with higher quality (e.g., the "good" rule \textit{Negation-NIL-Test-Rule}). Without testing additional rules, the rest of the plan can be solved resulting in the same partial derivation tree as known from the previous episode (compare Figures 2 and 4). This is a demonstration of how code templates from episodic instances and from generalization frames can play the role of additional preconditions to the rule stated in the episodic frame. These preconditions are satisfied if the code matches the datum template. Rules are preferred to other rules that, in other situations, would be preferred according to the list of sorted rules. What happens here can be described as the triggering of a "top-down" diagnostic process by data in a "bottom-up" fashion.

In the present case, however, the current plan and the code match exactly the code in the Datum slot and the plan in the Plan slot from the episodic frame NIL-TEST.TS.4-1. Therefore, the analysis of this part of the code does not have to be recomputed. Instead, this part of the resulting derivation tree can be reconstructed from episodic frames belonging to Episode No. 4. Preferring episodically indexed rules and reconstructing partial derivation trees (EPI) results in testing 49 rules (48%) less and applying four rules (30%) less without success as compared to diagnosing the same code without considering episodic information (NON-EPI, cf. Table 6).

\section*{AN EVALUATION STUDY}

Results from the example demonstrated in the previous section seem to show a clear advantage if the diagnostic process uses information from the learner model. However, two questions arise. First, one may wonder...
whether the example used above is more than an anecdote carefully chosen to demonstrate the apparent power of the model. It has to be shown for the full bandwidth of exercises from the LISP course and for different programmers that using information from the individual episodic learner model will reduce the computational effort in diagnosing code. Second, it has to be shown that learning individual episodic information will reduce the computational effort more than only learning how often rules were applied successfully in diagnosing problem solutions by a comparable group of users.

To answer these questions, the system was trained with data from a first group of novices learning LISP with ELM-PE. Afterwards, data from a second group of novices were submitted to evaluate the system's performance depending on the amount of heuristic information used during analyses to select rules.

Method

The same procedure was used in both the training phase and the evaluation phase. All completed function definitions coded as solutions to tasks from the exercises from the first six lessons in our introductory LISP course programmed in ELM-PE were collected as data. All function definitions presented as examples in the course materials were added as if they were solved by the learner in advance. In a simulation run for data from one subject, solutions to programming tasks were analyzed in the order in which they were produced by the subject when working with ELM-PE. Upon starting a new lesson, the examples from the course materials were analyzed and the episodic learner model was updated. These episodes served as prestored cases that were known to the programmers. Subsequently, function definitions produced as solutions to programming tasks were analyzed one after another. After each analysis, the individual learner model was updated with information from the derivation tree and frames were generalized as described in the previous sections. Additionally, the frequency count of every rule mentioned in the derivation tree was updated.

For all 14 runs in the training phase, the frequency counts of all rules were stored at the end of each simulation run and were preloaded in the following run. That is, the first run started with a random order of rules within the Sorted-Rules slot in each concept. In the following runs, rules were sorted according to frequency counts from the previous run. In the end, the system had learned which rules were used most often and most successfully when diagnosing data from a representative group of students. The frequency counts contribute to a canonical learner model that can be used to adapt the system to the behavior of novices learning LISP.

In the evaluation phase, data from another group of 14 subjects participating in the first six lessons of an introductory LISP course were used.
During the six lessons, subjects worked at 39 to 43 different programming tasks producing 47 to 119 function definitions that served as data for this simulation study. The mean number of evaluated function definitions (i.e., attempts to solutions of the programming tasks) ranged from 1.15 to 2.83. A lower value indicated that subjects needed fewer coding-evaluate-test cycles to produce a correct solution (or, in some cases, to finish the task without having reached a final correct solution). Over all, 1030 function definitions produced by subjects (in addition to the predefined function definitions from the examples of the lessons) were analyzed in the simulation runs.

Four different types of runs were performed. These four versions differed in the amount of information that was used to sort rules and in the amount of individual information considered during analyses.

**RANDOM (RAND):** In the RAND runs, rule instances in the Sorted-Rules slot of each concept were sorted randomly. In each of the five runs, a different random order applied with each subject getting the same five random orders. During analysis, no episodic information was used. This simulation type served as the control run.

**RANDOM + QUALITY (QUAL):** In the QUAL runs, rule instances in the Sorted-Rules slot of each concept were sorted randomly as in the RAND runs and then sorted into quality groups according to the rules' qualities ("good", "suboptimal", and "bug"). During analysis, no episodic information was used. That is, minimal information (rule quality) was used to trigger the order in which rules were tested.

**RANDOM + QUALITY + FREQUENCY (FREQ):** In the FREQ runs, rule instances in the Sorted-Rules slot of each concept were sorted as in the QUAL runs. After that, rule instances within each quality group were sorted according to the rules' frequencies as learned in the previous training phase. During analysis, no episodic information was used. That is, a canonical learner model was used to trigger the order in which rules were tested.

**EPISODE (EPI):** In the EPI runs, rule instances in the Sorted-Rules slot of each concept were sorted as in the FREQ runs. In these runs, an individual episodic learner model was built up. During analysis, rules preferred in similar previous episodes as indicated from episodic frames were used to trigger the diagnostic process. Rules that were applied successfully in previous episodes in a similar context were tested and applied first.

For every subject, five simulation runs with different initial random orders of rules were performed for all four evaluation types with all data. For each analysis of a function definition, statistics were collected indicating the computational effort to perform the analysis (e.g., how many rules were tested, how many rules were applied without final success, that is, how often backtracking was required, and how long the analyses lasted). For each of the 1030 analyses, results from the five simulation runs with different initial random orders of rule instances were averaged.
Results

Diagnoses' solutions were categorized into quality groups "good", "suboptimal", "bug", "unknown", and "new" according to the lowest quality of a rule reported in the diagnoses' derivation trees. 559 (54.3%) of all 1030 diagnoses resulted in an overall "good" quality, 22 (2.1%) were classified as "suboptimal" solutions, and 384 (37.3%) were classified as "bug." In 60 (5.8%) of all cases, the "default" rule applied in at least one branch of the derivation tree. That is, no other rule existed that could explain the corresponding part of the program code. In the EPI version, a "new" rule was created in this case. In the EPI runs, in 20 (1.9%) cases the solutions were classified as "new", that is, the "new" rule applied (instead of the "default" rule). In one third of the cases that could not be explained by rules that already existed in the knowledge base, students produced a solution that was so similar to a previous one that the new learned rule applied. That is, the system had learned successfully and it was able to give the feedback to the user that the same problem had been encountered in a previous situation. In 5 cases (0.5%), the diagnosis failed, that is, the process stopped without any result when a deadline of 2500 applied rules was reached.

The problem-solving effort in an analysis of program code can be measured in different ways. Both the number of rules tested and the number of rules applied without success can serve as an estimate of how strong the effect of different types of knowledge on the computational effort of a diagnosis may be. However, taking into account episodic knowledge requires the additional computational effort of searching for episodic frames and matching the current situation to information stored with episodic frames. Thus, the time needed to complete a diagnosis is a direct measure of how much a user will benefit from episodic learner modeling in practice. The number of rules needed to analyze the 1030 function definitions varied widely. In some cases, very unusual and complicated errors had to be explained that resulted in many logical transformations between different plans. This resulted in large variances and in a distinctive skew of the distributions of the number of rules tested and of diagnoses times. Therefore, the nonparametric Friedman test was used to evaluate simulation results. Means as well as medians and mean ranks for dependent variables from all four versions of simulation runs are shown in Table 7.

Tested Rules

The cognitive analysis of problem solutions tests rules sequentially in the order given in the Sorted-Rules slot from the concept frames until an explanation with an acceptable quality is found. In the case of considering frequency information from a canonical learner model and/or individual information from the episodic learner model, rules that were applied successfully in previous situations will be tested first, deviating from the sequence of rule
TABLE 7
Mean Problem Solving Effort

A) Number of Rules Tested

<table>
<thead>
<tr>
<th></th>
<th>R-N</th>
<th>RQ-N</th>
<th>RQF-N</th>
<th>RQF-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>610.5</td>
<td>370.2</td>
<td>334.0</td>
<td>248.3</td>
</tr>
<tr>
<td>Median</td>
<td>353.8</td>
<td>126.2</td>
<td>109.0</td>
<td>32.0</td>
</tr>
<tr>
<td>Mean Rank</td>
<td>3.890</td>
<td>2.623</td>
<td>2.298</td>
<td>1.189</td>
</tr>
</tbody>
</table>

B) Diagnosis times (seconds)

<table>
<thead>
<tr>
<th></th>
<th>R-N</th>
<th>RQ-N</th>
<th>RQF-N</th>
<th>RQF-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>17.165</td>
<td>9.695</td>
<td>8.864</td>
<td>6.952</td>
</tr>
<tr>
<td>Median</td>
<td>7.085</td>
<td>3.173</td>
<td>2.811</td>
<td>2.118</td>
</tr>
<tr>
<td>Mean Rank</td>
<td>3.785</td>
<td>2.500</td>
<td>1.987</td>
<td>1.728</td>
</tr>
</tbody>
</table>

instances in the Sorted-Rules slots. When an acceptable explanation is found, no further rules are tested even if they have higher priorities. Thus, the reduction of the number of rules tested during a diagnosis indicates how well information from different learner models—whether canonical or individual—triggers the diagnostic process. This does not mean that the computational effort will be drastically reduced, because in many cases rules may be excluded from application due to very simple tests from the rule premises or just because the first subplan from the consequences fails. The effect will be increased, however, if complete parts of an explanation structure can be reconstructed from information stored with episodic frames (e.g., the example shown in Figure 4).

To demonstrate how much the computational effort of an analysis was reduced, one must compare the number of rules tested when using episodic and/or frequency information to analyses without such heuristic information. In Table 7A, the mean and median numbers of tested rules and the mean ranks as reported from the Friedman test are shown. This test reports significant differences between treatments ($\chi^2 = 2356$; $df = 3$; $p < 0.001$). Individual comparisons show significant differences between all pairs of treatments (all differences between mean ranks ≥ diff$_{crit} = 0.178$; $df = 3$; $N = 1030$; $p < 0.001$). Grouping rule instances according to rule quality reduces the amount of rules tested compared to a pure random order. This effect increases if the system learned the frequencies of successful applications of rules in advance. However, the effect of considering individual episodic information during diagnoses on the number of rules tested is the largest effect observed in these analyses.
Rules Applied Without Success
The number of rules applied without success in all simulation runs highly correlated with the number of rules tested. Thus, these data are not reported separately. However, as the number of rules applied without success gives an impression of how much backtracking occurred during diagnoses, it is interesting to see in how many cases no backtracking occurred at all. From a total of 1030 cases, in 0, 98, 314, and 439 cases no backtracking occurred in the RAND, QUAL, FREQ, and EPI runs, respectively. That is, in the case of random order of rule instances (RAND) not a single case without backtracking was observed while in case of the canonical learner model (FREQ) in 30.5% of all cases no backtracking occurred and in case of considering the episodic learner model (EPI) in 42.6% of all cases the diagnosis of code followed a direct path without backtracking.

Time
While the number of tested rules indicate the potential effect of episodic information on the diagnostic process, the time needed to perform an analysis will be more meaningful in characterizing a system that is used on-line. One may question whether the additional computational effort needed to search for compatible episodic frames compensates or even reverses the benefits of episode-based diagnoses. Obviously, additional time is spent in creating episodic frames and storing and generalizing them in the episodic learner model. This updating process, however, is done in the background while the learner reads help messages or proceeds to work at the next task.

Compared to the mean time from the RAND runs, mean times from the EPI, FREQ, and QUAL runs are 43.5%, 48.4%, and 59.5% shorter, respectively (Table 7B). Due to the skewness of the time distribution, the median of times will give a more practical expectation of how long diagnoses will last in most cases. Median times from the EPI, FREQ, and QUAL runs are 55.2%, 60.3%, and 70.1% shorter compared to the median time from the RAND runs, respectively. The median time from the EPI runs is 24.7% shorter than from the FREQ runs. Comparing times with the Friedman test shows that these differences between conditions are significant ($\chi^2 = 1556$; $df = 3$; $p < 0.001$). Individual comparisons indicate significant differences between all pairs from the four treatments ($d_{crit} = 0.178$; $df = 3$; $N = 1030$; $p < 0.001$).

These measures of central tendency give only a rough impression of how treatments differ. For more practical purposes, that is, for diagnoses in online use, the distribution of time needed to complete diagnoses is more important. The number of diagnoses that can be completed within a given time bound will be a good estimation of how long a user has to wait for help in practice. Figure 5 plots the percentage of diagnoses completed by the four simulation versions (RAND, QUAL, FREQ, and EPI) within different time
Figure 5. Percentage of cases diagnosed by RAND, QUAL, FREQ, and EPI runs within different running time bounds. (A) Simple programming problems (Lessons 1 to 3), $N=454$. (B) List recursion problems (Lessons 4 to 5), $N=576$. 
bounds. The complexity of programming problems increases during lessons of the LISP course. Therefore, two different plots with different problem complexity are shown (Figure 5).

Figure 5A plots results from diagnoses of problems students worked at during Lessons 1 to 3. These problems, in most cases, are relatively simple. In the EPI runs, more than 50% of these problems could be diagnosed within less than one second, 90% within 5 seconds, and 95% within 8 seconds. Advantages of the "individual" EPI version over the "canonical" FREQ version can be seen especially for shorter diagnoses and diminish for longer lasting diagnoses. Figure 5B plots results from Lessons 4 to 6. In these lessons, function definitions solving list recursion problems had to be programmed. These problems are more complex than most problems in the three introductory lessons. In the EPI runs, almost 50% of all diagnoses could be completed within 3 seconds while less than 30% of all diagnoses in the FREQ runs could be solved within this time bound. The advantage of the EPI version persisted even for more complicated diagnoses lasting longer than 10 seconds. For these more complex problems, the RAND version performs very poorly due to a lack of any heuristic information about how to prefer rules during diagnosis. Within a time bound of 3 seconds, less than 1% of all problems could be solved, and even within a time bound of 17 seconds, less than 50% of all diagnoses were completed.

The main advantage of the case-based diagnosis (EPI) stems from reconstructing parts of the derivation tree when both the subplan and the corresponding part of code that is analyzed match the subplan and the code stored with an episodic frame, respectively. If the reconstruction mechanism is turned off and episodic frames only trigger rule selection, mean times in the EPI runs are 4.2% shorter than in the FREQ runs. The mean difference (0.373 seconds) is statistically significant (paired t-test, $t = 3.643, df = 1029, p < 0.001$).

**Discussion**

This simulation study answers the two questions asked initially whether the example shown in Table 6 is typical for the whole bandwidth of diagnoses and whether taking into account episodic information may outperform considering only frequency statistics of rules. First, the effect of episodic information demonstrated in the example appears not to be due to specific cases used to illustrate the model. To the contrary, medians of the number of rules tested from 1030 cases (Table 7A) show clearly that similar or even higher effects than observed from the example (Table 6) can be found in many cases.

Second, considering mostly information from episodic frames resulted in reducing the computational problem-solving effort of diagnoses as indicated by the number of rules tested and by the time needed to complete the
analysis (Table 7). It happened very rarely that episodic information
directed an analysis into a dead end that would not have been reached without
using this heuristic information. The strongest reduction effect stemmed
from reconstructing complete explanation subtrees from information stored
with episodic frames. This happened when the code and the plan from the
current situation matched completely the corresponding template from an
episodic frame. This effect was strongest when the episodic frame was an in-
stance of a higher order concept. Information from episodic frames, how-
ever, was restricted to the context in which it was used in a previous episode.
It could not spread or generalize to other concepts, even if concepts were
highly related.

Rule frequency, in contrast to episodic frames, does not contain any in-
formation about the situation in which the rule was applied successfully. As
explained in the previous section, code templates stored with episodic and
generalization frames play the role of additional preconditions to a rule that
applied in the corresponding previous episode and that are fulfilled in case
the episodic frame matches the current situation.

PREDICTING INDIVIDUAL PROBLEM SOLUTIONS

ELM is a case-based learning model that represents knowledge about a par-
ticular learner in terms of episodic information. If this learning model is
valid, then it should predict individually the learners' programming behavior.
This assumption was tested in a second simulation study. We automatically
predicted the program code that the 14 novice LISP programmers from the
first study produced as solutions to recursive programming problems during
three lessons. In this study, the individual episodic learner model was built
from explanation structures for the examples from the course materials and
from the learner's final solution to each programming task. All subjects
worked at all 17 different tasks.

For every new task subjects worked at, the system predicted an individual
solution in four different versions as in the first study. In the first run
(RAND), rule instances in the concepts’ Sorted-rules slot appeared in ran-
don order and the first applicable rule was chosen and applied to generate
code. In the second run (QUAL), rule instances were grouped according to
their quality, so that, in most cases, correct solutions were predicted. In the
third run (FREQ), rule instances were sorted according to frequencies of
successful applications in the learning phase (canonical learner model) as
described in Study 1, and in the fourth run (EPI), rules were selected indi-
vividually according to the episodic learner model built up during simula-
tion runs for each subject.

For each subject and for each task, we compared the first complete func-
tion definition the programmer coded to the function code generated and
predicted by the system. There were a total of 238 such situations. Table 8 reports the number of cases with correct predictions, that is, the code was predicted exactly (except for the name of the function and for names of local variables) or the code differed only in the sequence of arguments in commutative functions (e.g., equal and +).

In the RAND run, not a single prediction was correct. Therefore, these data are not reported in Table 8. In the QUAL, FREQ, and EPI runs, the code that the student produced as a first attempt was predicted correctly in 56, 112, and 122 cases, respectively. Differences in these frequencies are statistically significant (Cochran Test, \( Q = 98.6, df = 2, p < .001 \)). There is a drastic improvement in correct predictions when the system had learned which rules were most frequently applied in a training phase with a control group of students (FREQ). Correct predictions occurred even more often when considering information from the individual episodic case base (EPI). Frequencies of predictions from the FREQ and EPI runs (47.1% and 51.3%, respectively) are significantly different (McNemar Test, \( \chi^2 = 6.75, df = 1, p < .01 \)).

In all three different prediction versions, rule instances in the Sorted-Rules slots of the concepts were principally grouped according to their quality. Therefore, "good" rules had the greatest chance of being used when generating the predicted code. Another reason for preferring "good" rules when generating code in the EPI version stems from the high number of correct problem solutions by students. In 55.9% of all tasks, students coded a correct solution in the first trial that was diagnosed as a "good" solution. After debugging or improving their code, students finally came up with a "good" solution in four fifths of the other cases. Thus, in the end, for 90.8% of all tasks a "good" solution existed. Therefore, for most concepts used during diagnoses, episodic frames existed with at least one frame indicating how a "good" rule was used to solve a plan addressing this concept. Even if other frames existed that indexed rules with lower quality, rules were preferred in the order of their quality. Therefore, a more meaningful estimate of how well code is predicted is the percentage of solutions that were solved "well" by students in the first attempt and that were predicted

| Table 8: Number and Percentage of Correct Predictions for Program Code |
|------------------------|---------|---------|---------|
|                       | QUAL    | FREQ    | EPI     |
| Number of Predictions | 56      | 112     | 122     |
| Percentage of All Solutions (N=238) | 23.5%   | 47.1%   | 51.3%   |
| Percentage of "Good" Solutions (N=133) | 42.0%   | 84.2%   | 91.7%   |
correctly. In the FREQ and EPI versions, 84.2% and 91.7% of these 133 cases, respectively, were predicted correctly (see Table 8).

One of the problems with ELM is to predict errors. Very often it happens that a learner makes an error only once (in the beginning, when a new skill is learned). We often observed that the same error occurred in several attempts to solve the same task but was not produced again in the next task. However, some tasks later, or in another lesson, it sometimes happened that a previous error was produced again in the same context as before. This can be explained assuming that students try to solve a new task in analogy to a previous task—very often the task they just solved—changing as few elements as possible (Escott & McCalla, 1988; Weber and Bögelsack, 1995). Therefore, the system will be able to predict very well the algorithm the programmer follows in coding a new solution, but in most cases it will not be able to predict an error. From the point of view of a help system, this is not a disadvantage. The purpose of predicting code is to search for a previous solution or for a previous example that will be similar to an expected, individual solution to the current problem (Weber, 1991). In order to help the programmer, the system will present a correct solution and, therefore, a previous solution that is similar to an expected correct solution will serve as the best analog.

RELATION TO OTHER SYSTEMS

ELM has been introduced as a CBR-system that is used in the knowledge-based programming environment ELM-PE to diagnose and to explain solutions to programming problems offered by the users of the system. The diagnosis of program code is based on an explanation-based generalization procedure that is triggered by information from an individual episodic learner model which was built from derivation trees explaining previous problem solutions. ELM is related to different research areas, especially to research on learner and user models in intelligent tutoring systems and knowledge-based help systems and to research on case-based reasoning and learning. Several remarks about such relations have already been made in previous sections of this article. In the following section, these relations will be discussed in more detail.

Relation to Case-based Reasoning Systems

The diagnostic component of ELM-PE follows a case-based reasoning approach in that it relies on the episodic learner model ELM. ELM adapts the diagnostic process to an individual user of the system and retrieves individual programming episodes that can be used as analogs or remindings. In contrast to most other CBR-systems, ELM does not possess a prestored case
base. Cases are accumulated over time, starting with examples from the learning materials the learner studied from beginning to solve programming tasks. One can consider examples that are inserted first into the case base as some type of prestored cases. However, the processes of diagnosing these examples, of creating episodic cases, and of generalizing them follow the same mechanisms that are used later with individual solutions to programming tasks. In this sense, ELM represents a uniform, case-based learning system.

Initially, the diagnostic component cannot draw upon a wide variety of different cases. Thus, it must be able to solve the diagnostic problems completely without considering episodic information from stored cases. In this sense, ELM can be called a hybrid case-based reasoning system (Smith, Langston, & Nisbett, 1992). In principle, diagnosis in ELM is rule-based, but similar previous (partial) solutions can be used to shorten the diagnostic process. Additionally, rules can be preferred that were successfully applied in similar previous problem-solving situations. In many cases, this results in reducing the problem-solving effort.

In ELM, cases are indexed from concepts mentioned in plans that are followed during diagnosis. Rules stored with indexed cases can be preferred if the corresponding part of the code that is diagnosed matches the datum pattern stored with the case. Additionally, the context of the problem-solving situation expressed by the current plan has to correspond to the plan context from the episodic case. Such an indexing mechanism is similar to using plan transformations and failure rules as abstract indices (Owens, 1989). Pieces of cases are spread over the knowledge base in ELM. They are stored as episodic frames according to the concept frames from which they are instances. This allows indexing cases directly from concepts without being limited to single plan transformations and specific failure rules.

Explanation-based techniques are used in other CBR systems, too. For instance, explanation-based indexing is used in a system by Barletta and Mark (1988). However, in their system, cases are stored and retrieved as a whole. In ELM, an explanation-based technique is used to explain solutions to programming problems. The resulting explanation structure is distributed according to concepts from the knowledge base listed in the explanation structure. Such a distributed storage of cases can be found in the case-based parser DMAP (Martin, 1989; Riesbeck & Martin, 1986). In ELM as well as in DMAP, information stored in the case memory is accessed directly without a complex indexing mechanism. In ELM, partial solutions from previous cases are directly accessed from plans during diagnosis, while in DMAP, episodic structures are directly accessed from patterns that match the currently focused part of the input string.

ELM and DMAP differ in how they process new cases (i.e., new texts in DMAP and new programming solutions in ELM). When understanding new texts, it is not always obvious what the intention of a new message will
be. In DMAP, expectations can be set up from higher order goals existing in the system already, and from previous text passages, or they have to be built up when the new text is processed. Thus, expectations have to be created from patterns that match parts of the input, and many different hypotheses may exist in parallel at the same time. Although expectations represent top-down analysis of text, most of the text understanding works bottom-up. In ELM, diagnosing program code depends on a plan description of the programming task. Plans represent top-down expectations about how a solution to the programming task may be coded. These plans are refined step by step, until code is generated that will match the code being analyzed. This is purely top-down processing. Episodic cases, however, introduce bottom-up processing into the diagnostic process. Cases indexed from the code trigger the problem-solving process in that they prefer rules and transformations that were applied successfully in previous, similar situations.

Relation to User Models in Tutoring and Help Systems
The episodic learner model ELM is part of knowledge-based programming environment ELM-PE. Thus, its relation to knowledge-based components of other tutoring and help systems that support the acquisition of programming skills has to be discussed. Among the many tutoring and help systems developed during the last ten years are those supporting traditional programming languages. Most systems support LISP (e.g., the CMU-LISP tutor, Anderson, 1993; Anderson et al., 1989; Anderson & Reiser, 1985, and the SCENT advisor, McCalla & Greer, 1988, 1993), or PASCAL (e.g., the systems BRIDGE, Bonar & Cunningham, 1988, and PROUST, Johnson, 1986, 1988). Others are environments for graphical programming languages (e.g., the LISP-related graphical programming systems GIL, Merrill, Reiser, Beekelaar, & Hamid, 1992; Reiser, Beekelaar, Tyle, & Merrill, 1991, and GLUE, Waloszek, 1995, and the problem-solving monitor ABSYNT, Möbus & Schröder, 1993; Möbus, Schröder, & Thole, 1995). Relations to the CMU-LISP-tutor and to the SCENT-advisor that both support learning LISP will be discussed here in more detail.

Relation to the CMU-LISP Tutor
The CMU-LISP tutor is most similar to ELM. Both systems focus on supporting students who learn their first programming language. Both systems teach LISP as a pure functional programming language. Modeling a learner in the CMU-LISP tutor is based on Anderson's ACT theory (Anderson, 1983, 1987, 1993). Rules representing elementary programming skills are organized according to goals that control the learners' problem-solving process. In ELM, rules are assigned to concepts that represent elementary knowledge units. However, declarative knowledge in ACT does not directly correspond to knowledge structures in ELM. ELM describes procedural
knowledge needed to solve programming problems and to code programs. This can be described by rules without requiring declarative representational structures (as in most other cognitive systems, e.g., SOAR, Newell, 1990; Rosenbloom, Newell, & Laird, 1991).

The model-tracing approach in the CMU-LISP tutor interprets behavior observed from a particular learner in the scope of a model representing an ideal learner. Programming errors are identified by specific rules describing erroneous or suboptimal solutions with respect to the ideal learner. In ELM, the learners’ problem solutions are analyzed with respect to a tutor’s knowledge about how the programming task can be solved and about the type of typical errors and suboptimal solutions that can be found in novices’ programming. With increasing knowledge about a particular learner, ELM adapts to his or her programming style. It is not obvious whether both approaches are different in principle. Also, the model the tutor in ELM has about programming and about solving problems may or may not be considered analogous to the model of an ideal learner with suboptimal and bug rules describing deviations from the optimal problem-solving behavior. Rules in both systems do not differ in principle, although they may differ in the grain size describing problem-solving behavior. However, because the CMU-LISP tutor is based on ACT theory, the mechanism of activating facts in working memory and of selecting rules differs from selecting and applying rules in the CBR approach used in ELM.

Despite these correspondences and differences, the problem discussed with overlay models (Ohlsson, 1986) persists. It is not clear whether the learner models utilized in both systems actually represent the knowledge of a learner. And it can be doubted that concordance with and deviation from the model of an ideal or expert learner reflect the processes of knowledge and skill acquisition in learners. One solution to this problem would be to see the learner model as a model that a tutor builds up of a particular student.

Relation to the SCENT Advisor
The SCENT advisor is an intelligent advising system for novices learning to program recursive functions in LISP (McCalla & Greer, 1993). The central intelligent component in SCENT uses case-based and granularity-based reasoning methodologies. It is the task of this component to determine what strategies the learner has used when solving a programming task. A model-based approach is used to recognize such strategies in that it compares the learners’ solutions to models in a predefined strategy library. This strategy library is organized according to a static granularity hierarchy in which domain concepts are represented as objects that recognize instances of themselves in the learners’ solutions to a programming task. For example, a CDR-RECURSION object would recognize a cdr-recursion in a LISP program if it
contains instances of the concepts CDR-REDUCTION and NULL-BASE-TEST. That is, objects are described in terms of their components that are themselves objects. This is comparable to recursively imbedding plans used to match problem solutions in ELM.

This static hierarchy is instantiated for particular patterns that correspond to parts of the learners' solutions. It results in a flexible and robust hierarchical representation of different levels of programming strategies that occur in LISP programs. The instance hierarchy is used as a basis for creating cases in the sense of CBR (Kolodner, 1993). A full case consists of a task description, a solution in LISP, a granularity hierarchy of strategy instances, and a set of annotations on this hierarchy. These annotations are used to attach hints and advice to specific instances in the hierarchy. Advice can be called by the learner on demand. This is similar to the sequence of hints that are made available to the learner in ELM-PE. In contrast to ELM-PE, the learner can specify the level of detail of the advice offered in the advice window.

The SCENT advisor as well as ELM use case-based reasoning methods. However, both approaches differ significantly. ELM is a case-based learning system aggregating cases from observing a particular student and using information from previous cases to trigger the diagnostic process. SCENT, in contrast, has cases prestored by the developer of the system allowing for detailed advice tailored to a particular case on different granularity levels. Thus, both systems differ in that SCENT uses a categorization type CBR while ELM uses a problem-solving type CBR.

Another difference to ELM is that SCENT has not yet been included into a complete programming environment. Similar to PROUST (Johnson, 1986), it can only be used to analyze complete solutions to programming tasks. Therefore, incomplete solutions cannot be analyzed at all. Also, and in contrast to ELM-PE and to the CMU-LISP tutor, the SCENT system by its own is not able to support programmers during planning and coding functions. However, recent work has tied the PETAL program development environment into SCENT (Greer, McCalla, Price, & Holt, 1994).

As McCall and Greer (1993) report, many cases have to be added to the system in advance to allow for advice in a wide range of different solutions. SCENT does not store individual cases observed from a learner automatically. Thus, it does not adapt on-line to a particular learner. New cases observed from a learner can only be added to the system by the designers of the system. This allows the tailoring of specific advice that can be annotated to this case, but it depends on the intuition of the designers as to why the learner produced the observed code. Having human intervention in the selection of cases allows more knowledge to be inserted into the diagnostic process—namely the selection of typical cases that actually occur with real learners.
CONCLUSIONS

One of the main goals in developing ELM was to model problem solving and the acquisition of problem-solving skills in novices learning a new programming language. In two studies it could be shown that the system adapted to users by two different learning mechanisms. In a first learning step, ELM learned how frequently rules were applied successfully from diagnosing code that a group of programming novices produced to solve programming tasks in an introductory LISP course. These frequency data were used in following diagnoses to trigger rule selection. They served as a canonical learner model. In a second learning step, ELM learned during diagnoses of problem solutions from each particular user by collecting and storing cases that explained how problems were solved and which rules the user preferred and applied successfully in problem solving. Learning both general rule frequencies and individual episodic information resulted in two consequences that enable ELM to be used successfully in an intelligent learning environment: First, diagnoses were performed faster, and second, the system was able to predict individual problem solutions.

Reducing the problem solving effort of diagnoses in combination with shorter response times is a necessary prerequisite of a help system to be used on-line. Using information from the individual episodic learner model resulted in shortest diagnoses times with most diagnoses completed within less than 10 seconds even for more complex recursive problems. This is necessary for an on-line help system to be accepted by users. The strong effect of episodic information could be traced back mostly to reconstructing explanations structures from previous diagnoses. One could ask whether such chunking information can be put beforehand into the knowledge base in terms of rules that apply when the very specialized preconditions match. In principle, this could be done, but at least two disadvantages are bound up with compiling episodic chunks into fixed rules. On the one hand, many such specialized rules have to be implemented into the system because it is not known in advance which rules will really be used. This is especially true for more complex problems for which a lot of different solution paths and combinations of partial paths exist. Additionally, such a mechanism would limit the system to be used with a fixed set of tasks and programming concepts and would make it more difficult to scale it up. The current version of ELM benefits from its flexibility of using more general rules that can easily be adapted to new programming concepts and to new tasks.

Until now, adding new information to the system is very low level and can only be done by the developers of the system. Results from using analogical replay in PRODIGY (Veloso, 1994) show that a case-based reasoning system similar to case-based learning in ELM not only can be scaled up but also is one of the most promising attempts to reduce exponential explosion of the computational effort when problems become more complex. Our results from the first study reported in this article are in concordance with
Veloso's results and indicate that the system can be scaled up. Especially, episodic learner modeling will help to reduce the complexity of diagnoses when new concepts and rules are added.

On the other hand, specialized rules are lacking the flexibility that is a feature of information from episodic frames. Even without using chunk information, episodic frames can be used flexibly to trigger the selection of rules that results in faster diagnoses as well as in predicting code more accurately than a pure rule-based approach. The main advantage of ELM stems from combining episodic-based selection of rules similar to derivational analogy (Veloso & Carbonell, 1993) with replaying chunks similar to SOAR (Laird, Newell, & Rosenbloom, 1987). This is just what we observed from programmers in our studies, too. They reuse complex parts of previous solutions and they possess the flexibility (or have to learn it) to adapt old solutions to the new problem and to follow different paths, that is, to combine solutions from different similar previous problems. This is one reason why ELM is a cognitive model of how novices solve programming problems.

Predicting individual problem solutions successfully is not only useful for validating cognitive modeling in ELM. It is a prerequisite to present useful examples and reminders individually by the explanation-based retrieval mechanism (EBR) used in ELM-PE (Weber, 1991, 1995). In EBR, analogs for solving a new problem are found by first predicting a solution to the problem based on information of how the programmer usually solved similar tasks. This solution is diagnosed and stored temporarily into the episodic learner model. Then the most similar episodes are computed according to their organizational similarity (Wolstencroft, 1989). In an experiment we just finished, students using ELM-PE with complete functionality, that is running the episodic version of ELM in the background and the possibility to provide appropriate examples by the system, worked fastest through exercises and performed best in the final programming tasks.

In several studies using different versions of ELM-PE during the past few years (Weber & Möllenberg, 1994, 1995), students using a version with cognitive diagnosis solved most of the exercises correctly and performed best in the final tests. Feedback given in a final questionnaire confirmed our observations that student liked to get help from the cognitive diagnosis on demand and that they benefited from explanations given in case of errors or of suboptimal solutions. Thus, ELM is an adequate model, it is practically useful, and working in ELM-PE makes fun for students.

REFERENCES


