International Journal of Artificial Intelligence in Education (1999), 10, 238-256

Evaluation of a Constraint-Based Tutor for a Database Language

Antonija Mitrovic *Computer Science Department, University of Canterbury Private Bag 4800, Christchurch, New Zealand*
Email : tanja@cosc.canterbury.ac.nz, http://www.cosc.canterbury.ac.nz

Stellan Ohlsson *Department of Psychology, University of Illinois at Chicago*
Email : stellan@uic.edu, http://www.uic.edu/depts/psch/ohlson-1.html

Abstract: We propose a novel approach to intelligent tutoring in which feedback messages are associated with *constraints* on correct problem solution. The knowledge state of the student is represented by the constraints that he or she does and does not violate during problem solving. Constraint-based tutoring has been implemented in SQL-Tutor, an intelligent tutoring system for teaching the database query language SQL. Empirical evaluation shows that (a) students find the system easy to use, and (b) they do better on a subsequent classroom examination than peers without experience with the system. Furthermore, learning curves are smooth when plotted in terms of individual constraints, supporting the psychological appropriateness of the constraint construct.

INTRODUCTION

Individualized instruction requires a model of the learner's knowledge state and the ability to construct and update that model on line. Euphoria caused by initial successes with automated student modeling in the domains of arithmetic (Brown & Burton, 1978; Burton, 1982) and algebra (Sleeman et al., 1990; Sleeman et al., 1989) were soon dampened by the realization that the student modeling problem is intractable in its general form (Holt et al., 1994; Self, 1990; Ohlsson, 1986, 1991).

The key difficulty is that the formal knowledge representations that intelligent tutoring system (ITS) researchers have inherited from the field of artificial intelligence demand overly detailed and specific models of the student's knowledge. Whether a student model is encoded in Horn clauses, Lisp functions, rule sets or schemas, a mechanism for inferring a model of a particular student has to specify dozens -- sometimes hundreds -- of individual knowledge elements. If the student model is to be executable, these knowledge elements have to be as specific as program code. This level of specificity cannot be attained. It is impossible to know in such detail exactly what is in a student's head on the basis of in-depth interviews, let alone the incomplete and course-grained information about the learner that is available to an ITS. We refer to this as the overspecificity problem.

However, a student model can be useful even though it is not complete and accurate (Stern, Beck & Woolf, 1996). For example, empirical studies have shown that teachers use loose and incomplete models of students and yet are highly effective (Holt et al., 1994; Leinhardt & Ohlsson, 1990; Putnam, 1987). Consistent with this observation, successful student modeling techniques are compromises, designed to resolve the conflict between overspecificity and empirical underdetermination. Examples include model tracing (diagnose one step at a time rather than entire problem solutions; Anderson et al., 1990), Bayesian networks (estimate probabilities for a pre-defined set of knowledge elements; Martin & VanLehn, 1993) and fuzzy set modeling (Hawkes & Derry 1989/90). These techniques work either because they limit the specificity of the resulting models (Baysian nets, fuzzy sets) or because they restrict the instructional scenario in ways that significantly simplifies the modeling problem (model tracing).

The purpose of constraint-based modeling (CBM) is to overcome the overspecificity problem via abstraction (Ohlsson, 1992). The key idea is that knowledge about a domain can be represented by constraints on correct solutions in that domain. Each constraint indirectly represents a set of erroneous solutions, namely all solutions that violate that constraint. An expert model consists of a set of constraints that partitions problem solutions into acceptable and unacceptable in the same way as an expert. A student model consists of the set of constraints that he or she does and does not violate.

The constraint-based technique has been implemented in SQL-Tutor, an intelligent tutoring system for a database language. We first develop the principles of constraint based modeling. A description of SQL-Tutor is followed by an empirical evaluation in terms of usability, learning and effect on classroom performance. In particular, we show that constraint based modeling satifies the smooth curve criterion proposed by Anderson and co-workers (Anderson et al., 1995) as the sign of a psychologically appropriate knowledge representation.

CONSTRAINT-BASED MODELING

A constraint-based model represents knowledge about a domain as a set of constraints on correct solutions. The constraints select, out of the universe of all possible solutions, the set of correct solutions. More precisely: They partition the universe of possible solutions into the correct and the incorrect ones.

A Formalism for Constraints

Ohlsson and Rees (1991) introduced a formal notation for constraints. The unit of knowledge is called a *state constraint*. Each state constraint is an ordered pair $<C_r, C_s>$, where C_r, the *relevance condition,* identifies the class of problem states for which the constraint is relevant, and C_s, the *satisfaction condition*, identifies the class of (relevant) states in which the constraint is satisfied. Each member of the pair can be thought of as a set of features or properties of a problem state. Thus, the semantics of a constraint is: *if the properties C_r hold, then the properties C_s have to hold also* (or else something is wrong). A simple example from the domain of Lisp programming is the following constraint:

> *If the code for a Lisp function has N left parentheses, there has to be N right parentheses as well (or else there is an error).*

In this example, *the code has N right parentheses* is the relevance criterion *and the code has N left parentheses* is the satisfaction criterion. This example has the unusual feature that the relevance criterion is always satisfied, so the constraint is always relevant. (In the database domain to be discussed in the next section, approximately 20% of the constraints turn out to be relevant in every problem state.)

As a second example, consider the following constraint on the addition of fractions:

> *If (x+y)/d is given as the answer to x/d1 + y/d2, then it has to be the case that d=d1=d2 (or else there is an error).*

In more idiomatic English, this constraint says that you cannot add fractions by adding their numerators unless they have the same denominator. In this example, C_r, the relevance criterion, is the complex *predicate (x+y)/d is given as the answer to x/d1 + y/d2,* and C_s, the satisfaction criterion, is the predicate *d=d1=d2.*

A state constraint can be represented as a pair of *patterns,* where each pattern is a list (conjunction or disjunction) of elementary propositions. In this representation, each part of a constraint is analogous to the condition side of a production rule. Alternatively, state constraints can be implemented as pairs of (complex) Lisp predicates. The important point is that each state constraint is a pair of tests on problem states.

The computations required to test whether a given problem state is consistent with a set of constraints are straightforward: Compare the state against all constraints and notice any constraint violations. This is a two step process. In a first step, all the relevance patterns are tested against the problem state to identify those constraints that are relevant in that state. In a second step, the satisfaction patterns of the *relevant* constraints are tested against the problem state. If the satisfaction pattern of a relevant constraint matches the current state, then that constraint is satisfied. If the satisfaction pattern of a relevant constraint is not satisfied, then that state violates the constraint.

Instructional Application

The basic idea of constraint-based tutoring is to equip an instructional system with a set of constraints for the target domain and to inform the student on-line about his or her constraint violations. If the constraints are formulated in a psychologically appropriate way, the system will evaluate a student's solutions in the same way as a domain expert.

The state constraint approach circumvents the overspecificity problem by providing two pedagogically relevant forms of abstraction. First, a constraint base enables selective evaluation of problem solving steps. Not all problem solving steps are equally informative or important in diagnosing a student's knowledge. For example, in solving a problem in elementary arithmetic or algebra, the student will almost certainly type an equal sign somewhere in his or her answer. This step might in and of itself contain minimal information about the student's thoughts about the problem. Rather than trying to predict such a step (i.e., to model the generative knowledge that produced the step), an instructional system might be better off to wait to see what the student does next.

No additional mechanism needs to be implemented to allow a constraint-based system to ignore pedagogically uninformative steps. If the step does not evoke any constraint (i.e., does not cause any relevance condition to match that did not match in the previous state), then the step is *de facto* ignored. Constraints can be written so as to react only to problem states that do contain pedagogically significant information about the learner (Ohlsson, 1992).

Second, a constraint base circumvents the overspecificity problem by allowing an instructional system to operate with classes of *pedagogically equivalent* solution paths. The basic purpose of an instructional system is to map student performances onto instructional actions (e.g., typing out a particular instructional message). Hence, the system needs to group student solutions into classes of solutions that require the same instructional response from the system.

For example, consider the following general constraint for programming:

The code for an iterative routine must contain at least one GOTO statement that returns control to the first step in the iteration (or else the iteration is not coded correctly).

It does not matter by which sequence of programming steps the learner arrived at a program that violates this constraint. All sequences of steps that lead to such a violation require the same instructional response: Talk to the learner about the cyclic nature of iterative computations (Soloway & Spohrer, 1989). A constraint C implicitly defines a bundle of solution paths, namely all paths that pass through some problem state that violates C. If C is a pedagogically motivated constraint, all those paths should require the same instructional response.

A set of constraints does not have to be complete to be useful. An incomplete set might fail to capture certain rare errors, but as long as it captures the most common errors, it can still provide valuable feedback. In complex domains, to formally decide whether a set of constraints is complete, i.e., whether it will catch every error, is an intractable problem.

Discussion

The concept of state constraints was invented to solve a deep puzzle about skill acquisition: Human beings can catch themselves making errors. For example, skilled typists often make typing errors that they immediately correct. This ability forces a distinction between *generative* and *evaluative* knowledge (Norman, 1981; Ohlsson, 1996a). The function of generative knowledge (e.g., a rule set) is to produce actions vis-a-vis the current problem and the function of evaluative knowledge (a set of constraints) is to evalute action outcomes as desirable or undesirable.

This distinction suggests that the acquisition of a new cognitive skill consists, in part, of the transfer of knowledge from the evaluative to the generative component. A detailed statement of this theory is available in Ohlsson (1993, 1996a). Constraint-based computer models of skill acquisition in arithmetic and college chemistry are described in Ohlsson (1993), Ohlsson, Ernst and Rees (1992) and Ohlsson and Rees (1991).

Different learning theories have different implications for the design of ITSs (Ohlsson, 1991). The state constraint theory suggests that the knowledge base of a constraint-based tutoring system should contain the constraints that the student would have, had he or she already attained mastery of the target task. Hence, such a tutoring system plays the role of an amplified evaluative knowledge base. Our conjecture is that access to such a knowledge base will speed up and augment the transfer of information from the evaluative to the generative component.

This approach is quite different from attempting to model (rather than amplify) the student's generative (rather than evaluative) knowledge, the typical aim of most student modeling techniques. The instructional implications of the state constraint theory are discussed further in Ohlsson (1996b).

IMPLEMENTATION

SQL-Tutor is an intelligent tutoring system, designed and implemented by the first author, that helps students formulate queries in the Structured Query Language (Mitrovic, 1997, 1998). We first summarize the domain and the pedagogical problem it poses and then describe the components of SQL-Tutor.

Domain and Overview

Structured Query Language (SQL) is the dominant database language today (Elmasri & Navathe 1994). It is used for both interactive and programmed access to databases. At the elementary level, the main activity in using SQL is to formulate queries vis-à-vis a particular database.

To use SQL, students have to learn its syntax and semantics, how to develop queries and how to test and repair queries. Although SQL is a simple and well-structured language, students find it difficult to master. Many difficulties are due to the high memory load the students experience while defining queries. They have to keep in mind database schemas, names for attributes and tables, the semantics of the latter and the corresponding integrities. One symptom of the memory load is that incorrect queries often contain invalid table or attribute names.

Other errors are due to students' misconceptions about SQL in particular and the relational database model in general. Students find it particularly difficult to grasp the concepts of grouping and restricting grouping. So-called join conditions and the difference between aggregate and scalar functions are other sources of confusion. These observations are consistent with the reports of other researchers (e.g., Kearns, Shead & Fekete, 1997).

SQL is commonly taught in classrooms. Solutions to query formulation problems are demonstrated on a blackboard and complemented by laboratory exercises in which SQL queries are shipped to the relevant relational database management system (RDBMS). However, direct interactions with an RDBMS are pedagogically problematic. First, if a student's solution is syntactically correct but semantically wrong, the RDBMS will accept it, generate the resulting table and present it to the student. The majority of students do not analyze the result of a query in depth and mistakenly come to believe that their solution to the problem was successful.

Second, even if a query is syntactically incorrect and hence triggers an error message, the feedback produced by the RDBMS is typically difficult to comprehend. In our experience, it is almost impossible for students to learn from such feedback. For example, Figure 1 illustrates a situation in which a student is required to specify a SELECT statement with five clauses to search a database of movies for directors that have won more than one award for their comedies. The ideal solution is based on a single table (consider only comedies), a single grouping operation (form a group for each director) and a HAVING clause (consider only groups where the number of awards is greater than unity). However, the student used two tables, which makes the query inefficient because the join operation specified in the FROM clause is time consuming. Furthermore, the student did not realize that summary information was needed for each director and he failed to specify the GROUP BY and HAVING clauses.

Figure 1 shows the error message generated by the relevant RDBMS (which in this case was Ingres). As the reader can verify in the Figure, this message is uninformative and would be of little use to most students. This is the pedagogical situation that SQL-Tutor was designed to improve upon.

Figure 1 also shows how SQL-Tutor would handle the situation discussed previously. The messages provided by SQL-Tutor take both syntactic and semantic aspects of the student's solution into account. For example, the first message is derived from the observation that the student specified two tables in the FROM clause, even though only one of them (MOVIE) is needed. This kind of feedback is crucial for effective learning in this domain.

Example: For each director, list the director's number and the total number of awards won by comedies he or she directed if that number is greater than 1.

```
Correct solution:                     Student's solution:

SELECT DIRECTOR,SUM(AAWON)            SELECT DIRECTOR,SUM(AAWON)
FROM MOVIE                            FROM DIRECTOR JOIN MOVIE
WHERE TYPE='comedy'                   ON DIRECTOR=DIRECTOR.NUMBER
GROUP BY DIRECTOR                     WHERE TYPE='comedy'
HAVING SUM(AAWON)>1
```

INGRES: E_USOB63 line 1, The columns in the SELECT clause must be contained in the GROUP BY clause.

SQL-Tutor:

- You do not need all the tables you specified in FROM!
- You need to specify the GROUP BY clause! The problem requires summary information.
- Specify the HAVING clause as well! Not all groups produced by the GROUP BY clause are relevant in this problem.
- If there are aggregate functions in the SELECT clause, and the GROUP BY clause is empty, then SELECT must consists of aggregate functions only.

Figure 1. A student error, RDBMS feedback and potential feedback from SQL-Tutor

SQL-Tutor is not intended to replace classroom instruction, but to complement it by providing a supportive problem solving environment. The system assumes that students have been exposed to the basic concepts of databases in lectures and that they are familiar with both the relational data model and the basics of the SQL language.

Because classroom observations indicate that students find query formulation quite difficult, the primary purpose of the current version of SQL-Tutor is to help students formulate SELECT statements. Although this focus is narrow, the techniques used to teach SELECT statements could be used to teach other types of SQL statements and many of the concepts needed to understand SELECT statements are essential for understanding other database languages as well.

SQL-Tutor is implemented in Allegro Common Lisp on SUN workstations and PC compatibles. See Mitrovic (1997, 1998) for implementation details. The system contains definitions of several databases, implemented on the RDBMS used in the laboratory where the system is employed. New databases can easily be added to SQL-Tutor by supplying the same

SQL files used to create the database in the RDBMS. SQL-Tutor also contains a set of problems for specified databases and the ideal solutions to them.

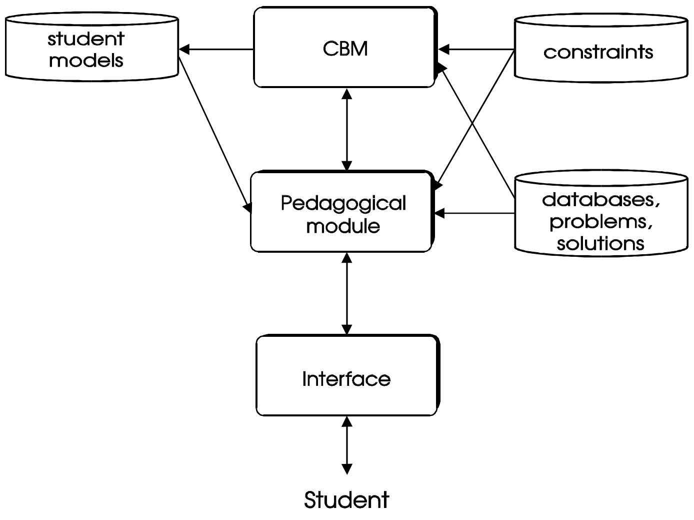


Figure 2. The architecture of SQL-Tutor.

The system consists of an interface, a knowledge base, a student modeler and a pedagogical module that determines the timing and content of pedagogical actions (Figure 2). The following subsections discuss the individual components in more detail.

Interface

At the beginning of a session with SQL-Tutor, a student is required to enter his/her name, which is used to retrieve the appropriate student model. If the student is logging on SQL-Tutor for the first time, a new student model is created and an initial screen gives information about how to use the system. The system provides help about specific aspects of the system via help menus and tool tips.

The main window of SQL-Tutor is divided into three areas, which are always visible to the student. The upper part of the window displays the text of the current problem and the student can always look there to remind himself/herself of the elements requested in the query. The middle part lists the clauses of the SELECT statement, thus exhibiting the goal structure of the task. The lowest part displays the schema of the currently chosen database.

The interface reduces memory load by displaying the database schema and the text of a problem, by exhibiting the basic structure of the query and also by providing explanations of the elements of SQL. Users can easily access the descriptions of databases, tables or attributes as well as the descriptions of various SQL constructs.

SQL-Tutor supports problem solving by exhibiting the relevant subgoals, i.e., the elements (clauses) of an SQL query. Students need not remember the exact keywords and the relative order of the clauses. They can obtain short descriptions of the functions and roles of the various clauses by selecting the appropriate clause or by asking for help from the main menu. Displaying the database schema is particularly important, because of the constant need to remember table and attribute names and their semantics. By easing the cognitive load involved in checking low-level syntactic details, the system allows students to focus on the higher-level aspects of query definition.

SQL-Tutor does not interrupt the student while he or she works on a query. Analysis of the student's solution begins when he or she clicks on the *submit* button. When the student submits his or her solution for evaluation, the pedagogical module sends it to the student modeler, which analyzes the solution, identifies mistakes (if there are any) and updates the student model appropriately. If the solution contains errors, the pedagogical module presents an appropriate feedback message.

Knowledge Base

The version of SQL-Tutor described in this article contains 406 constraints. The constraints form an unordered set. They were formulated by the first author on the basis of two sources of information: analyses of the target domain (Elmasri & Navathe 1994) and comparative analyses of correct and incorrect student solutions recorded while teaching the college course in computer science in which SQL-Tutor will eventually be employed. All constraints are general in the sense that their conditions can be tested against any problem.

The constraint notation in SQL-Tutor is expressive enough so that the system can test subtle features of student solutions and compare them to correct solutions. Relevance and satisfaction patterns can be arbitrary logical formulas, containing any number of atomic predicates. Some conditions are patterns that match parts of the student's solution or the ideal solution, while others are Lisp predicates. In addition to the relevance and satisfaction conditions, each constraint is associated with a number, a natural language description and the name of the clause to which the constraint applies.

We distinguish between two types of constraints. Constraints of the first type represent syntactic properties of queries. They refer only to the student's solution. Constraints of the second type represent semantic properties of queries. They operate on the relation between the student's solution and the ideal solution.

Syntactic Constraints

By syntactic constraints we mean constraints on the form a query. Constraints of this type are typically simple and easy to formulate. As an example, consider Constraint 2, which specifies that the SELECT clause of a SQL query cannot be empty (Figure 3). Unlike most constraints, Constraint 2 is always relevant, so its relevance condition reduces to "*t*", the Lisp symbol for a condition that is always satisfied. In the current constraint base, 76 constraints (19%) are always relevant.

```
(p 2
"The SELECT clause is a mandatory one. Specify the
 attributes/expressions to retrieve from the database."
 t
 (not (null (select-clause ss)))
 "SELECT")
```

Figure 3. The parts of Constraint 2.

The satisfaction condition of Constraint 2 verifies that the SELECT clause of the student solution (represented by the variable *ss*) is not empty. This is a straightforward combination of atomic Lisp predicates. The first part of the constraint is the verbal description that is included in the instructional feedback message that SQL-Tutor exhibits in case the constraint is violated. The last part of the constraint is the name of the clause that the constraint refers to. It, too, is included in any instructional message.

```
(p 146
 "You have used some names in FROM that are not from this database!"
 (bind-all '?n (find-names ss 'from) bindings)
 (or (attribute-in-db (find-schema (current-database *student*)) '?n)
     (valid-table (find-schema (current-database *student*)) '?n))
 "FROM")
```

Figure 4. The parts of Constraint 146.

Not all syntactic constraints are this simple. For example, Constraint 146 checks that all the names used in the FROM clause of a SELECT statement are valid names of attributes or tables in the currently selected database (Figure 4). In this case, the relevance condition verifies that the student has used at least some names in the FROM clause. If he or she has not, then this particular constraint does not apply (but others might). The satisfaction condition then tests that each of those names are to be found among either the relevant attributes or the valid tables for the database the student is currently working in. In this case, the tests contain no pattern elements, only complex Lisp predicates.

As a final example of a syntactic constraint, Constraint 22 checks whether the student is using the BETWEEN predicate correctly (Figure 5). Its relevance condition verifies that the WHERE clause is specified, and then finds all parts of that clause which are based on BETWEEN. Next, the satisfaction part of the constraint checks that each such condition is specified on a valid attribute, checks for the use of NOT within the condition, checks whether the AND keyword separates the lower and upper value of the interval and, finally, checks that the constants used are of the appropriate type. In this case, several conditions are of the pattern matching type, marked by the use of the "match" function.

```
(p 22
 "BETWEEN requires two constants of the same type as the
   attribute used in the condition, separated by AND."
 (and (not (null (where ss)))
      (member "BETWEEN" (where ss) :test ’equalp)
      (bind-all ’?a (names (where ss)) bindings)
      (match ’(?*d1 ?a ??n "BETWEEN" ?*d2) (where ss) bindings))
 (and (attribute-in-from ss ’?a)
      (member ’?n ’(nil "NOT") :test ’equalp)
      (match ’(?v1 "AND" ?v2 ?*d3) ’?d2 bindings)
      (equalp (find-type ’?a) (find-type ’?v1))
      (equalp (find-type ’?a) (find-type ’?v2))
 "WHERE")
```

Figure 5. The parts of Constraint 22.

Semantic constraints

By semantic constraints we mean constraints that have to do with the meaning of the symbols and commands in a query. Constraints of this type are typically more complex than syntactic constraints. Of course, the distinction between the two kinds of constraints is not strict and some constraints inspect both the syntax and the semantics of the student’s solution. For example, Constraint 361 verifies that if the results of a query are sorted in decreasing order in the ideal solution, the student’s solution uses the same ordering (Figure 6).

```
(p 361
 "Check whether you should have ascending or descending order!"
 (and (not (null (order-by is)))(not (null (order-by ss)))
      (bind-all ’?n (names (order-by ss)) bindings)
      (match ’(?*d1 ?n "DESC" ?*d2) (order-by ss) bindings)
      (not (qualified-name ’?n)))
 (match ’(?*d3 ?n "DESC" ?*d4) (order-by is) bindings)
"ORDER BY")
```

Figure 6. The parts of Constraint 361.

Discussion

The constraints are deliberately written to be highly modular. Each constraint focuses on one aspect of the solution, even in those cases when it would have been possible to cover both aspects in a single constraint. This implementation strategy increases the total number of constraints, but it allows us to attach a single instructional message to each constraint, which simplifies student modeling and the delivery of instruction.

Constraint-based modeling does not, in principle, require an ideal solution. However, in the SQL domain, such solutions were readily available and they allowed us to formulate certain constraints, particularly semantic constraints, which would have been much more difficult to formulate if the system had not had access to ideal solutions.

Student Modeler

When SQL-Tutor is initialized, the constraints are compiled into two structures, called the relevance and satisfaction networks, which resemble RETE networks (see Mitrovic, 1997, for technical details). There are three types of nodes in these structures: input, test and output nodes. The difference to RETE networks is that test nodes have a single input each, so the structures are trees, not unrestricted networks. Nevertheless, we will refer to these structures as "networks" for consistency with the terminology typically used in discussing RETE and other pattern matching techniques.

Constraint violations are identified by inspecting the student’s solution and by comparing it to the stored ideal solution. This is a two-step process. In the first step, the student’s solution and the corresponding ideal solution are propagated through the relevance network. The result is a list of constraints the relevance conditions of which match the current situation. In the SQL domain, the number of relevant constraints per student solution varies from 86 for the simplest query to well over a hundred for more complex queries.

In the second step, the satisfaction components of constraints whose relevance conditions match the current situation are compared to the current problem state. That is, the student’s solution and the ideal solution are propagated through the satisfaction network. If a satisfaction condition matches the state, the corresponding constraint is satisfied. The system takes no action. If the constraint is violated, this outcome is recorded. The student model consists of the list of violated constraints.

The student modeler records the history of each constraint. This record contains information about how often the constraint was relevant for the ideal solution to the practice problems the student attempted, how often it was relevant for the student’s solution and how often it was satisfied or violated. This information is accumulated in three indicators, called *relevant*, *used* and *correct*. This record is used by the pedagogical module.

Pedagogical Module

The pedagogical module generates feedback messages and selects practice problems. The instruction is individualized in the sense that both types of actions are based on the student model.

Feedback

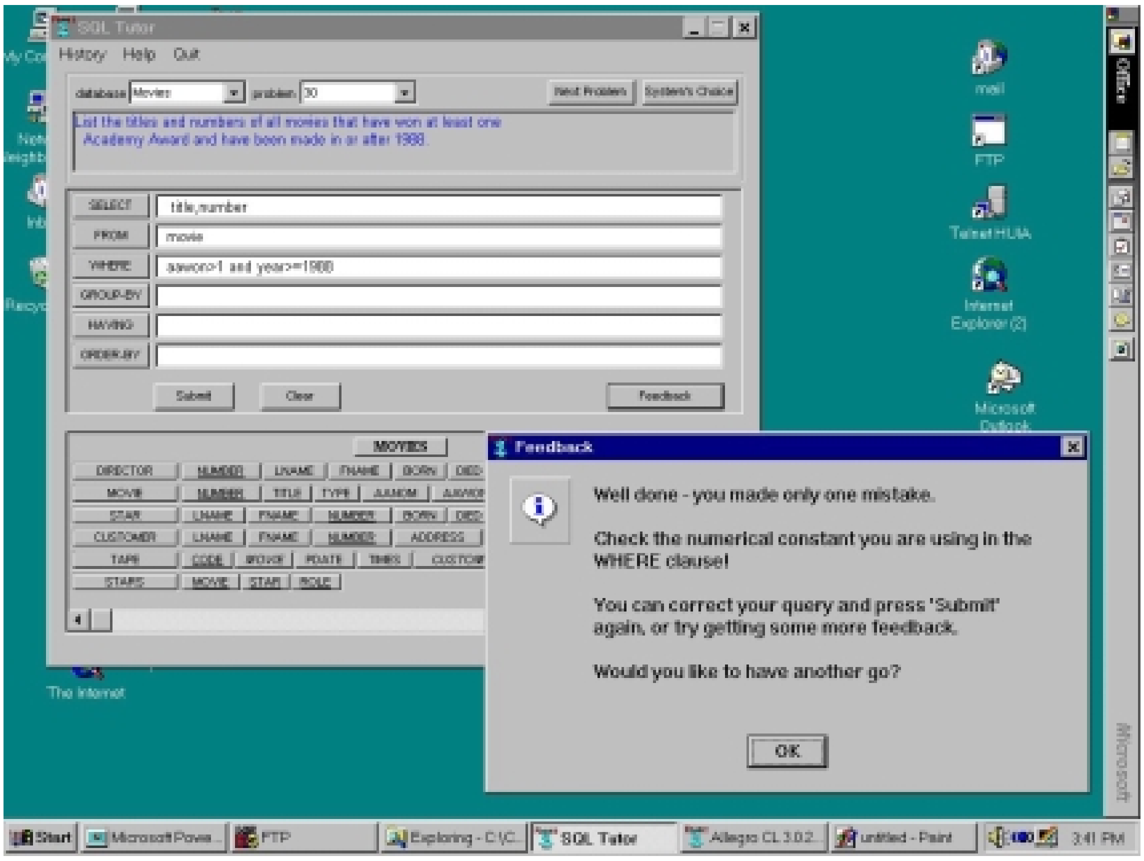
Unlike ITSs that use the model tracing technique (Anderson et al., 1990), SQL-Tutor does not follow the student step by step and it does not give feedback after individual problem solving steps. Instead, SQL-Tutor postpones evaluation and feedback until the student submits his or her solution. This is appropriate in this domain, because the order of the steps taken while formulating an SQL query is not constitutive of a correct or successful query.

A student’s solution to a query problem can violate several constraints. This is illustrated in Figure 1, in which the student’s solution violates four constraints. In such cases, SQL-Tutor examines all violated constraints and targets one of them for instruction. SQL-Tutor consults the history of each violated constraint and selects the constraint with the largest number of violations, computed as the difference between the *used* and *correct* indicators. The rationale for this rule is that if the student has violated the same constraint several times, then it is appropriate to target that constraint for instruction.

The student is told the total number of errors in his or her solution, but is only given feedback about one of them. The implicit assumption behind this practice is that it is easier for the student to work on one error at a time and that multiple feedback messages related to multiple errors or misconceptions might be confusing or overwhelming.

Students can receive feedback on incomplete solutions as well. There are constraints that check that mandatory parts of a SELECT statement are specified and there are also constraints that compare the student's solution to the ideal one. These constraints enable SQL-Tutor to provide feedback even when the student submits an empty solution, i.e., a solution that does not specify any part of the SELECT statement.

Feedback messages can vary in the amount of information they provide. In the current version of SQL-Tutor, there are five levels of detail: right/wrong, error flag, hint, partial solution and complete solution. At the least detailed level, a *right/wrong* feedback message informs the student whether his or her solution is correct or not. If there are errors, the student is told how many errors there are. An *error flag* informs the student about the query clause in which the error occurred. A *hint* gives more information about the type of error, as shown in Figure 7. In this case, the student is given a general description of the error, taken from the definition of the constraint. A *partial solution* displays the correct content of the crucial clause, while a *complete solution* displays the correct content of each clause in the query.



database: Movies problem: 30

List the titles and numbers of all movies that have won at least one Academy Award and have been made in or after 1988.

SELECT: title,number
FROM: movie
WHERE: aawon>1 and year>=1988
GROUP-BY:
HAVING:
ORDER-BY:

Well done - you made only one mistake.

Check the numerical constant you are using in the WHERE clause!

You can correct your query and press 'Submit' again, or try getting some more feedback.

Would you like to have another go?

Figure 7. An SQL-Tutor feedback message at the hint level.

The level of feedback is adjusted in the following way. When a student starts working on a new problem, he or she receives only feedback of the *right/wrong* type. If the student goes through several unsuccessful solution attempts, the feedback is upgraded to the *error flag* level and then to the *hint* level. The system never volunteers more than a hint, but the student can ask for partial and complete solutions by clicking on a *feedback* button and selecting the desired level.

Problem selection

Students can work their way through a prespecified sequence of problems by selecting the *next problem* option. They can also select a practice problem directly from a menu of problems. This feature allows students to go back and redo a problem they have already attempted but abandoned. The main pedagogical disadvantage with student-determined problem selection is that it has unpredictable effects on constraint coverage.

Students can also turn problem selection over to the system by selecting the *system's choice* option. SQL-Tutor then examines the student model and selects the next problem on the basis of two rules. First, it uses the number of violations of each constraint to identify a constraint that the student has yet to learn and then finds a problem for which that constraint is relevant, thus giving the student a chance to receive instruction about it. Second, the system can also identify constraints that have not been relevant for any of the problems that the student has attempted so far, and select a problem for which that constraint is relevant.

EMPIRICAL EVALUATION

The evaluation study was carried out in the Computer Science department at the University of Canterbury in New Zealand. The students were in their senior year. They had listened to six lectures about SQL and they all had at least eight hours of hands-on experience of query definition. The instructor (the first author) asked the students to volunteer for the evaluation study. Participation was anonymous. Out of the 49 students enrolled in the course, twenty choose to participate.

The students used SQL-Tutor in a two-hour session. The number of problems attempted varied between 4 and 26, with an average of 13.5. The average number of successfully solved practice problems was 11.5.

There were four observers present during the session. The observers agreed among themselves that the students were quite interested in interacting with the system and exploring its various functions. Although the system has been developed for individual learning, some students collaborated by comparing the feedback they received from the system while working on the same problems and by exchanging comments and explanations. Some students tested the abilities of the system by approaching the same practice problem in different ways.

All students' actions were recorded and the students filled out a questionnaire at the end of the session. We evaluate SQL-Tutor on three dimensions: usability, learning and effect on subsequent classroom performance.

Results 1: Usability

The students who participated in the study filled out a questionnaire (see Appendix A). All students were already familiar with SQL, either through lectures (25%) or through lectures plus some hands-on experience (70%). No student reported extensive previous experience of SQL.

The students were comfortable with the interface. The majority (75%) reported that they needed less than 5 minutes to start using the system; two reported that they needed 10 minutes; two students needed 30 minutes; finally, one student reported spending most of the two hours getting familiar with the system. When asked to rate ease of use on a scale from 1 (not easy) to 5 (very easy), half the students choose alternative 4. When asked whether the display of the query schema was understandable, 85% of the students choose alternative 5 on the same rating scale. When asked whether they enjoyed working with the system, 50% of the students once again choose alternative 4. In short, a majority of the students found the interface easy to learn, comprehensible and enjoyable.

Nevertheless, 65% of the students reported some software problems. These were of diverse kinds, including a few programming errors. Two students (10%) complained about the speed of the interface. (The interface of the Solaris version of the system, the version used in the study, is written in Tcl/Tk. It is slower than the interface for the PC version, especially when the system is running on older computers. Future implementation efforts will be directed towards improving the PC version.)

We also asked students to evaluate efficiency of learning. When asked to rate how much they learned from working with the system on a scale from 1 (nothing) to 5 (very much), the average rating was 2.9. When asked whether one hour with SQL-Tutor was equal to one hour of lectures and laboratory exercises with respect to learning, half the students agreed and half disagreed. One explanation for the relatively low values on these variables is that many of the students had already encountered the relevant databases and problems in their prior laboratory exercises. However, it is easy to add new databases and problems to SQL-Tutor.

The students rated the feedback messages as helpful on a scale from 1 to 5 and the average answer was 3.3. When asked whether they wanted more detailed feedback messages, 55% of the students said "yes" and 20% said "no." (There were a few suggestions on how to provide additional useful information, such as connecting the system to a DBMS, so that queries can actually be run and results inspected.) In short, the students felt that they learned something, but perhaps no more so than from other types of instruction, and the feedback messages were less helpful than they could have been for a significant minority of the students.

When asked whether they would like to work more with the system, 85% said "yes" and when asked whether they would recommend the system to another student, 75% said "yes." Consistent with these findings, we observed that the participating students continued to use the system on their own after the study. Furthermore, students who did not participate in the study also began using the system, presumably on a recommendation from those who did participate.

Results 2: Learning

Objective evidence of learning consists of significant change over time in one or more performance measures. Figure 8 shows the number of errors (constraint violations) per solution attempt, averaged across subjects, as a function of the number of solution attempts. (We use solution attempt as the unit, not solutions, because subjects did not always succeed in solving a practice problem they attempted, and they were free to return to a previously attempted problem.) There is a slight drop over the first part of the curve, but the trend is weak and overshadowed by the huge variations from trial to trial. The power law fit is very poor. Plotted in this way, the data provide little evidence that the students learned anything from their interactions with SQL-Tutor.

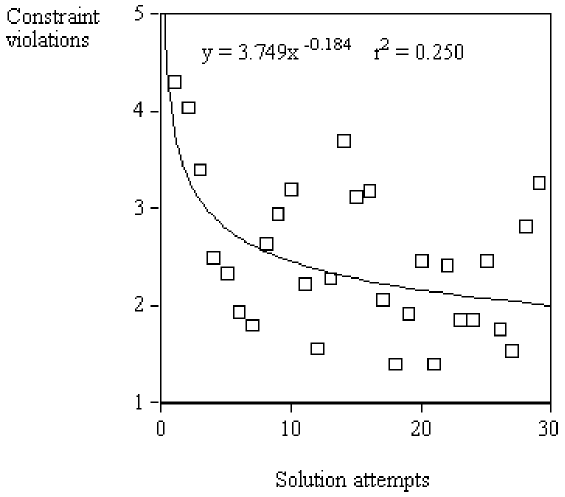


Figure 8. The number of constraint violations as a function of the number of practice problems attempted, averaged over subjects.

However, the research team led by John R. Anderson, Albert Corbett and Kenneth Koedinger at Carnegie-Mellon University has pointed out that plotting learning results in terms of phenomenological[1] units--e.g., time to complete a practice problems or correctness of a solution, number of practice problems--is theoretically unwarranted. In instructional settings, as opposed to laboratory experiments, practice problems differ in exactly which knowledge units they require for successful solution. Also, the students typically have some control over which practice problems they attempt, so two students who both have solved N practice problems will not have had the same training history. In this type of learning scenario, there is no reason to expect all learners to acquire the same knowledge. Consistent with this argument, learning curves plotted in terms of phenomenological units are often highly irregular.

In contrast, if we plot the speed or correctness by which a *particular knowledge unit* (e.g., a production rule) is applied as a function of the amount of practice *on that particular knowledge unit* (rule), then learning is a smooth, negatively accelerated curve that closely approximates a so-called power law (Anderson, 1993, Figs. 2.2, 2.3; Anderson & Lebiere, 1998, Figs. 2.1, 2.2). It is not the amount of practice on the target skill as a whole but the amount of practice per knowledge unit that determines the level of mastery (of that unit). This fact is consistent with the idea that knowledge units are learned one by one, independently of each other, and that the acquisition of any one unit is a regular process.

In SQL-Tutor, knowledge is represented in terms of constraints. If those constraints represent psychologically appropriate units of knowledge, then learning should follow a smooth curve when plotted in terms of constraints. To evaluate this expectation, we randomly selected 100 constraints among those constraints that were relevant at least once during the study[2]. For each of the selected constraints, the problem states in which that constraint was relevant were identified in each student's record and rank ordered from 1 through R. We refer to these as *occasions of application*. For each occasion, it was recorded whether the relevant constraint was violated or satisfied. This analysis was repeated for each subject.

From this transformation of the computerized records we can compute two variables of interest. The first is the probability of violating a given constraint C. To estimate this quantity, we computed, for each subject, the proportion of the 100 selected constraints that he or she violated on the first occasion of application, the second occasion, and so on. These proportions were averaged across the twenty subjects and plotted as a function of the number of occasions when C was relevant. The results are shown in Figure 9.

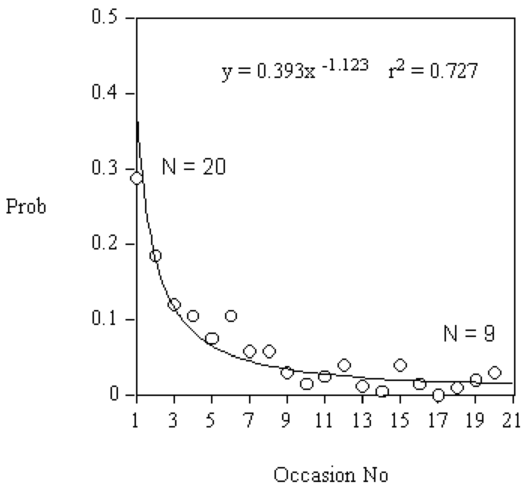


Figure 9. Probability of violating a constraint as a function of number of occasions when that constraint was relevant, averaged over subjects

As the figure shows, the relation between the probability of a constraint violation and the amount of constraint-specific practice is quite regular. The initial probability of violating a constraint is approximately 30%. (The fact that the students already had eight hours of practice on query definition before interacting with SQL-Tutor explains why this number is not higher.) After 10-12 encounters with problem states in which the constraint was relevant, the probability of a violation has decreased to a few percent. The decrease is quite orderly. The r^2 power law fit is .73, more than double that of the curve in Figure 8. Thus, the data support the belief that the constraint base parses the target knowledge into psychologically relevant units that are learned independently of each other and that the degree of mastery of a given unit is a function of the amount of practice on that unit.

The second quantity of interest that can be extracted from the computer records is the rate of mastery of the target skill as a whole. To estimate this quantity, we calculated the proportion of subjects who had zero constraint violations on each successive occasion of application. Figure 10 shows this quantity plotted as a function of occasion of application. There is a steep initial increase in the proportion of subjects who do not make errors. As the subject group approaches mastery, the curve fluctuates. This is a statistical artifact, caused by the fact that the data points for the higher values on the x-axis refer to fewer subjects. (The subjects were free to choose which as well as how many practice problems they attempted, so different students had different amounts of practice.) The logarithmic curve provides the best fit to these data with an r^2 fit of .89, once again a regular relationship.

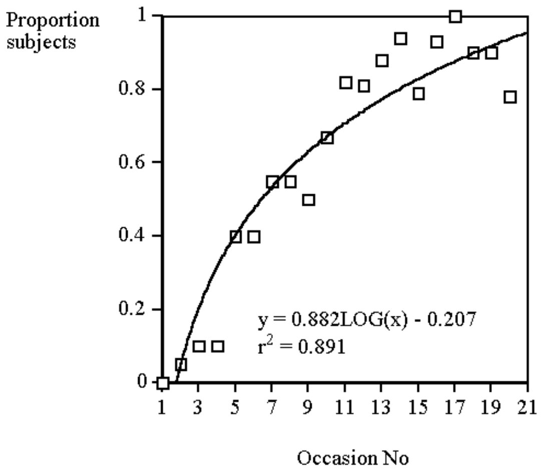


Figure 10. Proportion of subjects with zero constraint violations as a function of occasion to violate

In summary, the probability of violating a constraint decreased in a negatively accelerated fashion with increasing number of opportunities to acquire the knowledge embedded in that constraint. Furthermore, the proportion of subjects who violate no constraints increases smoothly and rapidly across occasions of applicability. Both analyses verify that the students learned something from their interactions with SQL-Tutor. The fact that the learning curve looks smooth when plotted in terms of the individual constraints, but not when plotted in terms of practice problems, provides a measure of support for the appropriateness of the constraint representation.

Results 3: Classroom performance

Because not all students in the course participated in the study, we can compare the achievement of those who participated and those who did not with respect to their scores on a subsequent examination. There were 49 students enrolled in the class; 3 of those did not take the final examination. Of the remaining 46 students, 20 participated in the study. Because the system was available after the study to all students whether they had participated in the study or not, the 26 non-participating students may not represent a pure control group.

The examination was conducted two weeks after the study. The students were asked to answer six query formulation problems of the same type as the practice problems posed by SQL-Tutor. The students had 90 minutes to complete the six problems. Their solutions were scored on a scale from 0-100 by the first author, using a blind scoring procedure.

The test score for the students who used the system was 82.7, while the corresponding score for those who did not was 71.2. The students who used the system scored, on the average, 11.5 points better on the examination than those who did not. This difference is statistically significant ($t= 2.68$, $p = .01$).

The standard deviation for both groups combined is 15.4. Hence, an increase of 11.5 points in the mean score is somewhat less than the one-sigma increase in performance observed for some other ITSs (Anderson et al, 1990, 1995). However, the size of the effect has to be judged against the fact that the students only used SQL-Tutor for a total of two hours in a semester-long course. In those studies that have exhibited one-sigma improvements, the students have typically used the relevant system throughout the entire semester (Anderson et al., 1995).

From this perspective, the improvement is larger than one would expect. One possible explanation for this is that the students were free to access SQL-Tutor after the study was over and several of them did so. Hence, the observed improvement is a function of the training during the study and however many hours the students used the system afterwards on a voluntary basis. We do not have objective records of how much students used the system after the study.

Summary of Results

The empirical evaluation showed that SQL-Tutor is user-friendly, that the students judged the feedback as helpful and the interface as easy to learn. Furthermore, the performance records show that that the students' performance increased significantly in terms of lower probability of violating specific constraints with increasing number of occasions to practice that constraint. Finally, those students who participated in the study performed significantly better than those who did not on a subsequent classroom examination. These conclusions should be regarded with caution in light of the fact that the participants in the treatment group were self-selected. Further evaluation studies that remedy this problem are currently under way.

DISCUSSION

The intractability of the student modeling problem is caused, in part, by the knowledge representations that have been used in ITSs to date. Horn clauses, Lisp functions and production rules were invented to encode executable programs. But the incomplete and fuzzy information available to an instructional computer system cannot support detailed and precise inferences about the student's cognitive strategy.

In contrast, the state constraint representation is not designed to encode executable programs. Although we use the "if-then" construct in English transcriptions of state constraint formulas, the state constraint idea cannot be absorbed either into the material implication of logic-based programming or the rule construct of production system architectures. A state constraint is a piece of evaluative knowledge. It is a tool for passing judgment, not for computing new results or inferring new conclusions.

Representing student behavior in terms of constraints provides two pedagogically useful forms of abstraction. First, problem solving steps that do not trigger constraints are effectively ignored. Hence, constraints allow the system to be selective in which aspects of student behavior it attends to. Second, each constraint represents a bundle of solution paths, namely all solutions that violate that constraint. All those solutions indicate a need to teach the knowledge embedded in the violated constraint.

Relations to Alternative Approaches

Constraint-based student modeling differs in significant respects from other approaches. Although each constraint encodes a piece of correct domain knowledge, a constraint base is nevertheless not an expert or ideal student model. It cannot be executed to solve problems. Because it does not require an executable expert model, constraint-based tutoring is applicable in domains in which such a model would be difficult to construct.

Because each constraint encodes piece of correct knowledge, a constraint base is not a bug library. A student is not represented by the incorrect knowledge he or she has, but by his or her constraint violations, regardless of whether these arise from the lack of correct knowledge or from erroneous knowledge. Consequently, it is not necessary to conduct extensive empirical research to identify and explicitly codify students' bugs (but it is, of course, necessary to conduct an in-depth task analysis).

A set of constraints nevertheless supports some of the functions of a bug library. A constraint violation provides the same information as a positive match between a theoretical bug description and a student behavior. The set of all possible violations of a constraint base represents the same universe of behaviors as the set of all positive matches to a bug library, but the representation is implicit rather than explicit.

Another advantage of constraint-based student modeling over the bug library technique is that bug libraries do not transfer well between different populations of students (Payne & Squibb, 1990). A constraint-base, on the other hand, encodes correct domain knowledge, which of course is the same across student populations.

Yet another strength of CBM is that it can recognize a correct solution submitted by the student, even if that solution is different from the ideal solution. If no constraint is violated, then the student's solution is correct with respect to the notion of correctness embodied in the constraint base. Hence, CBM need not be thrown off track by correct but creative or unusual solutions, a common problem with other student modeling techniques.

Potential Problems

In past work, we identified four potential limitations and problems with CBM (Ohlsson, 1992). In this section, we relate those difficulties to our experiences with SQL-Tutor.

First, it is not certain that pedagogically appropriate constraints can be identified in each and every domain. In the past, we have found the representation natural for arithmetic and chemistry (Ohlsson, 1993, 1996a; Ohlsson, Ernst & Rees, 1992; Ohlsson & Rees, 1991), but how general is this finding? We do not know. Task domains vary in multiple ways and along multiple dimensions and there is no way to answer this question theoretically. However, we encountered no principled difficulties in representing knowledge of the SQL database language in the form of constraints.

Second, even if appropriate constraints can be found, they might provide too loose a net so that too many student errors go unnoticed by the tutor. Our experience with SQL-Tutor has not confirmed this worry. The current version of SQL-Tutor contains approximately four hundred constraints. It is likely that this number will grow, as we consider new problem types and observe more students. However, these constraints allow the system to deal with all types of pedagogical situations encountered to date in the database query domain. SQL-Tutor does, in fact, select appropriate problems and generate appropriate feedback messages.

Third, we originally hypothesized that CBM is limited to domains in which the purpose of student modeling is to judge the correctness of successive problem solving steps (Ohlsson, 1992). Our experience in implementing SQL-Tutor has disproved this idea. SQL-Tutor system does not follow the student step by step, but carries out diagnosis with respect to his or her final solution. This proved to be very effective. We now believe that constraint-based diagnosis on final solutions will work in any domain in which those solutions have a rich internal structure.

A fourth worry mentioned in Ohlsson (1992) is that the acquisition of constraints might turn out to be no easier than the acquisition of expert rules or bug libraries. Knowledge acquisition is a slow, time consuming and labour intensive process. Anderson (1995) reports that the induction of a single production rule can require ten or more hours of work. Expert system researchers report that they can identify 2-5 production rules per day by interviewing domain experts. In contrast, each constraint in SQL-Tutor required an average of 1.1 hours of work, a significant saving. This may be a consequence of the fact that the same person served as both domain expert, knowledge engineer and the system developer, but may also be due to the appropriateness of the state constraint formalism.

Future Directions

Although we have emphasized the strengths of the state constraint idea, we do not claim that it provides the final solution to the problem of student modeling and hence to intelligent tutoring. On the contrary, we regard CBM as one technique among others, with its own unique profile of strengths and weaknesses that is likely to work well in some cases but not in others. Further experiments have to be conducted to reveal what those strengths and weaknesses are.

The general lesson of CBM is that progress towards the goal of intelligent, automated tutoring requires the invention of novel knowledge representations especially designed for this purpose. There is no reason to limit knowledge representation to the traditional representations, nor is there any reason to believe that the state constraint representation will remain the last word. We have as yet explored a mere fraction of the universe of possible knowledge representations.

Acknowledgments

The preparation of this paper was supported, in part, by research grant No U6242 from the University of Canterbury to the first author, and, in part, by award EIA 9720351 from the Learning and Intelligent Systems program of the National Science Foundation (NSF). We thank Kendrick Elsey and Andrew King for implementing the interface.

References

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

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Erlbaum.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, 42, 7-49.

Anderson, J. R, Corbett, A. T., Koedinger, K. R., Pelletier, R. (1995). Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4(2), 167-207.

Brown, J. S., & Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2, 155-192.

Burton, R. (1982). Diagnosing bugs in a simple procedural skill. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 17-183). New York, NY: Academic Press.

Elmasri, R., & Navathe, S.B. (1994). *Fundamentals of database systems*. Redwood, CA: Benjamin Cummings.

Hawkes, L. W., & Derry, S. J. (1989/90). Error diagnosis and fuzzy reasoning techniques for intelligent tutoring systems. *Journal of Artificial Intelligence in Education*, 1, 43-56.

Holt, P., Dubs, S., Jones, M., Greer, J.E. (1994). The state of student modelling. In J. Greer & G. McCalla (Eds.), *Student modeling: The key to individualized knowledge-based instruction* (3-35). Berlin: Springer-Verlag.

Kearns, R., Shead, S., & Fekete, A. (1997). A teaching system for SQL. *Proc. of the Australasian Computer Science Education Conference* (pp. 224-231). ACM Press.

Leinhardt, G., & Ohlsson, S (1990). Tutorials on the structure of tutoring from teachers. *Journal of Artificial Intelligence in Education*, 2, 21-46.

Martin, J., & VanLehn, K. (1993). OLAE: Progress toward a multi-activity, Baysian student modeler. In S. Brna, S. Ohlsson & H. Pain (Eds.), *Artificial intelligence in education: Proceedings of AIED93 (410-417).* Charlottesville, VA: AACE.

Mitrovic, A. (1997). *SQL-Tutor: a preliminary report* (Technical Report No. TR-COSC 08.97). Christchurch, New Zealand: Computer Science Department, University of Canterbury.

Mitrovic, A. (1998). A knowledge-based teaching system for SQL. In T. Ottmann & I. Tomek (Eds.), *Proceedings of ED-MEDIA '98* (pp. 1027-1032). Charlottesville, VA: AACE.

Norman, D. (1981). Categorization of action slips. *Psychological Review*, 88, 1-15.

Ohlsson, S. (1986). Some principles of intelligent tutoring. *Instructional Science*, 14, 293-326.

Ohlsson, S. (1991) System hacking meets learning theory: Reflections on the goals and standards of research in Artificial Intelligence and education. *Journal of Artificial Intelligence in Education*, 2(3), 5-18.

Ohlsson, S. (1992) Constraint-based student modeling. *Journal of Artificial Intelligence and Education*, 3(4), 429-447.

Ohlsson, S. (1993) The interaction between knowledge and practice in the acquisition of cognitive skills. In A. Meyrowitz and S. Chipman (Eds.), *Foundations of knowledge acquisition: Cognitive models of complex learning* (pp. 147-208). Norwell, MA: Kluwer.

Ohlsson, S. (1996a). Learning from performance errors. *Psychological Review*, 103, pp. 241-262.

Ohlsson, S. (1996b). Learning from error and the design of task environments. *International Journal of Educational Research*, *25*(5), 419-448.

Ohlsson, S., Ernst, A., & Rees, E. (1992) The cognitive complexity of doing and learning arithmetic. *Journal for Research in Mathematics Education*, 23(5), 441-467.

Ohlsson, S. & Rees, E. (1991). The function of conceptual understanding in the learning of arithmetic procedures. *Cognition & Instruction*, 8, 103-179.

Payne, S., Squibb, H. (1990). Algebra mal-rules and cognitive accounts of errors. *Cognitive Science*, 14, 445-481.

Putnam, R. T. (1987). Structuring and adjusting content for students: A study of live and simulated tutoring of addition. *American Educational Research Journal*, 24, 13-48.

Self, J. A. (1990). Bypassing the intractable problem of student modeling. In C. Frasson & G. Gauthier (Eds.), *Intelligent tutoring systems: At the crossroads of artificial intelligence and education* (pp. 107-123). Norwood, NJ: Ablex.

Sleeman, D., Hirsch, H., Ellery, I., & Kim, I.-Y. (1990). Extending domain theories: Two case studies in student modeling. *Machine Learning*, 5, 11-37.

Sleeman, D., Kelly, A. E., Martinak, R., Ward, R. D., & Moore, J. L. (1989). Studies of diagnosis and remediation with high school algebra students. *Cognitive Science*, 13, 551-568.

Soloway, E., & Spohrer, J. (1989). *Studying the novice programmer*. Hillsdale, NJ: Erlbaum.

Stern, M., Beck, J., & Woolf, B.P. (1996). Adaptation of problem presentation and feedback in an intelligent mathematics tutor. In C. Frasson, G. Gauthier and A. Lesgold (Eds.), *Intelligent tutoring systems* (pp. 603-613). New York: Springer-Verlag.

Footnotes

1. The term "phenomenological" is used here in the sense used by physicists, i.e., as referring to categories that are directly given in experience, rather than in the sense used by philosophers, i. e., as referring to a particular epistemology.

2. Because different constraints are relevant for different problems, a small group of constraints were never relevant for the particular practice problems the students attempted. We excluded those from the analysis.

APPENDIX A: USER QUESTIONNAIRE

1. What is your previous experience with SQL?
   a) only lectures   b) lectures plus some work   c) extensive use

2. How much time did you need to learn about the system itself and its functions?
   a) most of the session   b) 30 minutes   c) less than 5 minutes

3. How much did you learn about SQL from using the system? (circle only one)
   Nothing   Very much
   1 2 3 4 5

4. One hour with SQL-Tutor is more valuable than one hour of lectures or labs.
   a) agree   b) disagree

5. Did you enjoy learning with SQL-Tutor? (please circle only one and specify the reasons)
   Not at all   Very much
   1 2 3 4 5

6. Would you like to use SQL-Tutor more? (please circle only one and specify the reasons)
   a) Yes   b) Do not know   c) No

7. Would you recommend SQL-Tutor to other students?
   a) Yes   b) Do not know   c) No

8. Do you find the interface easy to use? (please circle only one and specify the reasons)
   Not at all   Very much
   1   2   3   4   5

9. Do you find the display of the schema understandable?
   a) Yes   b) Do not know   c) No

   Please specify the reasons:

10. Do you find help messages useful? (please circle only one and specify the reasons)
   Not at all   Very much
   1   2   3   4   5

11. Would you prefer more details in messages?
   a) Yes   b) Do not know   c) No

12. What do you like in particular about SQL-Tutor?

13. Is there anything you found frustrating about the system?

14. How can the interface be improved for you?

15. How can SQL-Tutor be improved for you?

16. Did you encounter any software problems or crashes?
   a) Yes   b) No