INTELLIGENT TUTORING SYSTEMS:

THE PRACTICAL IMPLEMENTATION

OF CONSTRAINT-BASED MODELLING

A thesis

submitted in partial fulfilment

of the requirements for the degree

of

Doctor of Philosophy in Computer Science

in the

University of Canterbury

by

Brent. I. Martin

University of Canterbury

2001

Acknowledgements

Thank you to my supervisor Dr Tanja Mitrovic, for her invaluable guidance during my research and for making this period of study so enjoyable. Thanks also to my associate supervisor Professor Ian Witten from Waikato University, who taught me how to write and provided valuable feedback on this thesis. I am also grateful to the other members of the Intelligent Computer Tutoring Group—especially Pramudi, Mike, Kurt, Amali and Konstantin—for the many fruitful discussions on research in progress. Thank you also to Jane Mackenzie for her help with the Language Builder tutor, and the Canterbury University COSC programmers for keeping the machines well oiled. I also appreciate the staff (especially Ray Bygate) and year seven students at Akaroa Area School for their enthusiastic evaluation of the Language Builder ITS.

Finally, I thank my wife Suky and son Hugh for enduring the inevitable financial constraints, for supporting me during this time, and for taking an active interest in my research.

This research was supported by a University of Canterbury Doctoral Scholarship.

Abstract

An Intelligent Tutoring System (ITS) differs from other educational systems because it uses knowledge to guide the pedagogical process. It attempts to optimise the student's mastery of domain knowledge by controlling the introduction of new problems, concepts and instruction/feedback. Central to this process is the student model, which provides information about what the student knows. The state of the art in student modelling is model tracing, which compares student actions against an "ideal" procedure.

Constraint-based modelling is a new domain and student modelling method that describes only pedagogically informative *states*, rather than following the procedure the student used to arrive at their answer. Ohlsson introduced the idea, which is based on learning from performance errors, but did not provide details of how it should be implemented. Even his definition of constraints is very broad. SQL-Tutor is an existing ITS that uses a constraint-based model. The representation of constraints within this system is as loose as Ohlsson's description. The constraints in SQL-Tutor are LISP code fragments, where domain structural knowledge is incorporated into the constraints via ad hoc functions.

In this thesis we present a more specific representation for constraints that obviates the need for complex user-defined functions. Constraints (and their associated taxonomies and domain-specific functions) are specified as pattern matches. This new approach has two advantages: the constraints are simpler to author, and they can be used to generate solutions on demand. We have used the new representation to create algorithms for solving problems and correcting student mistakes, and for generating novel problems to present to the student. We present the details of these algorithms and the results of both laboratory and classroom evaluations. The solution generation algorithm is demonstrated in laboratory testing to

be practical, and the problem generation algorithm, together with a new problem selection method, exhibits improved learning performance in the classroom.

We also present the design and implementation of an authoring system for constraint-based tutors and demonstrate its efficacy in authoring tutors for two domains. One of these, a tutor for English language skills, was evaluated in an elementary school classroom. This evaluation was a success. The students enjoyed using the tutor, found the interface easy to use, and felt that they had learned a lot. An analysis of their mastery of the constraints suggested that they did indeed learn the underlying principles in the course of the session. The authoring tool enabled us to develop this system quickly using a spelling resource book as the source of both the domain taxonomy from which to produce the problems (i.e. a vocabulary of words to use) and the principles for the constraints. The authoring tool provided all other functions. This evaluation therefore showed that our authoring tool allows the rapid creation of an effective ITS.

Table of Contents

List of Tables

List of Figures

1 Introduction

Intelligent Tutoring Systems (ITS) differ from classic computer-aided instruction (CAI) in the way they adapt to users' individual needs. This is accomplished by modelling what the student does or does not understand. The basis of this *student model* is a domain model, which is a detailed description of the subject being taught. A common element of an ITS is its provision of a scaffolded environment for the student to practise the skill they are trying to learn. The domain and student models may be used to provide detailed feedback on student answers, select new problems, and indicate to the user their current strengths and weaknesses. They are an effective way to teach students, and gains in the order of 1 to 2 standard deviations in performance are possible when compared with classroom teaching alone (Bloom 1984; Anderson, Corbett, Koedinger and Pelletier 1995). Teaching by ITS promises to be more efficient than one-on-one tutoring, although not necessarily as effective. Unfortunately, building them is hard and this imposes a major bottleneck in their use (Murray 1997).

Constraint-Based Modelling (CBM) (Ohlsson 1994) is an effective approach that simplifies the building of domain models. However, CBM is still a young approach that lacks detail. In this thesis we investigate how to build effective ITSs using CBM. We present a representation for CBM that is easy to use and facilitates automatic problem solving. We then demonstrate how it can be used to decrease the effort required to build an ITS by automatically providing detailed, student-specific feedback, and by generating new problems according to students' needs. The remainder of this chapter introduces ITS and CBM, describes our thesis, and outlines the structure of the remainder of this document.

1.1 Intelligent tutoring systems

In the early 1970s, Intelligent Tutoring Systems began to evolve from simple computer-aided instruction (CAI). In simple CAI the interface is static with respect to each user. Information is presented in a lecture (or "storyboard") fashion, grouped into topics to form some sort of curriculum. The student navigates their way through the curriculum according to their needs, however each student is presented with exactly the same information and choices. They may also be asked questions either on request or automatically, to test their understanding so far. Feedback on their answers is usually restricted to an indication of whether their answer was right or wrong, and what the correct answer was. If any further feedback is required, such as comments on individual incorrect answers, it must be handcrafted for each question.

The problem with such systems is two-fold. First, the information they present does not target their audience. Although the student may select parts of the curriculum they are interested in, this is performed at a very high level, and the actual content of each topic is unvarying. The system may therefore present information that the student is already familiar with, requiring them to wade through it in search of the parts that are of use. Worse, it may make assumptions about what the student knows, even though they have not covered the required part of the curriculum. The student will then need to hunt for the relevant concepts in the rest of the material.

This problem extends to the setting of exercises. On conclusion of a topic, a simple CAI often poses some questions so the student can see how well they have understood the material. However, the system may make invalid assumptions about what the student knows at this point, and hence set problems that they are unable to solve. Also, if the student has understood most of the content but is struggling with a particular aspect, the system is unaware of this and may not set any/enough exercises in the problem area.

Second, the feedback on problems is of limited use. People learn by applying the relevant skills, and so problem solving is an important part of learning. However, the usefulness of performing exercises is dependant on how much can be learned from mistakes made (Ohlsson 1996). To be helpful, the system needs to tell the student *why* the answer was wrong. In simple CAI, this is difficult, because the system has no

understanding of the domain: it simply presents information and problems that have been stored by a teacher. Any additional feedback is developed from scratch for each problem.

Early CAI adopted an approach called “linear programming”, where the topic was presented in very small steps, such that questions posed have at least a 95% chance of being answered correctly (Last 1979). CAI has since evolved into ITS (and other methods) via a series of improvements, which have deepened the level of adaptivity. Some examples are (O'Shea and Self 1983):

- Branching, e.g. (Ayscough 1977) – the program adapts its response depending on the answer given. For example, it might present corrective feedback for a given error, or engage in a dialogue;
- Generative (Palmer and Oldehoeft 1975) – generate new problems of appropriate difficulty for the student, according to their current performance;
- Simulation, e.g. (McKenzie 1977) – the student interacts with a “virtual laboratory”;
- Games
- Dialogue systems (Carbonell 1970) – an extension of branching CAI where the student interacts with the system in a natural language

Intelligent Tutoring Systems (ITS) have evolved from these early attempts. They are an example of *adaptive* educational systems. Adaptivity is an important extension of CAI. Instead of presenting static information, adaptive systems use domain knowledge to actively decide what to show the student next. Techniques such as active hypermedia (Brusilovsky 2000; Murray, Piemonte, Khan, Shen and Condit 2000) combine and format content for presentation, depending on what the student has so far seen and understood. Intelligent coaches (Lajoie and Lesgold 1992) tailor the interface of online “coaches” so that the help they provide is useful without being extraneous. Practice-based systems select problem tasks based on the students’ current understanding. Some systems combine aspects of all three approaches. A key attribute of ITS is that the adaptive aspects of the system are separated from the course content. In other words delivery of the course material is supported by features that facilitate adaptivity, such as a domain and student model, teaching strategy, etc.

1.2 The domain and student models

The benefits of ITS over standard CAI are a result of their adaptivity, which in turn is derived from their deep modelling. ITSs contain two main models: a domain model and a student model.

The domain model represents the subject being taught in such a way that the system can use it for reasoning. There are many possible representations, including semantic networks, production rules and constraints. What representation is adopted depends partly on how it will be used. It supports other functions such as information selection and representation, problem selection, and feedback generation.

Whereas the domain model is common to all users of the system, the student model varies between students, or groups of them. It is a representation of their beliefs. This may take many forms, including general measures such as level of competence, rate of acquisition, attentiveness and motivation. Commonly, it includes detailed information such as which parts of the curriculum the student has visited, what problems they have solved and not solved, and, ideally, which concepts they have grasped or failed to grasp. The student model provides the ITS with adaptability. Given the system's current state plus the information from the student model, decisions will be made about how next to proceed. Because the student model is included, behaviour will be unique to that student.

The student model is usually related in some sense to the domain model. One common approach is to use an overlay: the student model is a kind of "window" to the domain model, providing a unique view of the underlying domain concepts coloured by the student's beliefs. As a simple example, it may specify that each individual knowledge unit has been learned or not learned. When talking about the student model, it is therefore not usually possible to separate it from the domain model, or, conversely, the representation of the domain model usually characterises much of the student model.

1.3 Constraint based modelling

CBM is a method that arose from experiments in learning from performance errors (Ohlsson 1996). Ohlsson proposes that we often make mistakes when performing a task, even when we have been taught the correct way to do it. He asserts that this is because the declarative knowledge we have learned has not been internalised in our procedural knowledge, and so the number of decisions we must make while performing the procedure is sufficiently large that we make mistakes. By practicing the task however, and catching ourselves (or being caught by a mentor) making mistakes, we modify our procedure to incorporate the appropriate rule that we have violated. Over time we internalise all of the declarative knowledge about the task, and so the number of mistakes we make is reduced.

Some domain model methods such as model-tracing (Anderson, Corbett, Koedinger and Pelletier 1995) check whether or not the student is performing correctly by comparing the student's procedure directly with one or more "correct" ones. In CBM, we are not interested in what the student has done, but in what *state* they are currently in. As long as the student never reaches a state that is known to be wrong, they are free to perform whatever actions they please. The domain model is therefore a collection of state descriptions of the form:

> *"If <relevance condition> is true, then <satisfaction condition> had better also be true, otherwise something has gone wrong."*

In other words, if the student solution falls into the state defined by the relevance condition, it must also be in the state defined by the satisfaction condition.

SQL-Tutor (Mitrovic 1998) is an example of an ITS that uses CBM. The domain model consists of over 500 constraints. A simple overlay student model is used, which records the number of times each constraint has been satisfied or violated. Although we have built a new tutor using the methods described in this thesis (see Section 7.2.5), we used SQL-Tutor as the basis of much of this research. This is chiefly because it already contains a model for a rich and complex domain, the SQL database language. This enabled us to test our ideas thoroughly without needing to build a new

domain model of similar complexity from scratch, which is difficult and time-consuming; instead we modified the existing domain model to fit our new approach. Further, it provided us with a full working system on which we could test individual ideas. Finally, SQL-Tutor had already been subjected to four evaluations between 1998 and 2000, providing a wealth of student performance information that could be used as input for our testing.

1.4 Limitations of CBM

In his definition of CBM, Ohlsson does not include implementation. In particular, the domain model is limited purely to describing how to critique a student solution. Even the student model representation is left for further research. SQL-Tutor, for example, uses a very simplistic student model that does not include any form of curriculum. CBM is also not concerned with how problems are produced or selected, and provides detail of only one type of feedback: declarative messages that are attached to each constraint.

Many ITSs contain a problem solver, whose function is to determine the correct solution to a given problem state, or, at the very least, the next best action to take. Because a constraint-based model contains all the required information to determine whether or not a solution is in a valid state, we propose that this is sufficient to solve the problem. Further, we contend that because CBM always considers the student solution's state, it is capable of correcting any student solution. We argue that this is useful because it allows the system to show the student how to eliminate their mistakes without misleading them by introducing unnecessary changes.

CBM's modular nature allows information relating to different domain concepts to be mixed together at will, subject to satisfying the constraints. This provides an opportunity for automatic experimentation within the domain: SQL-Tutor could potentially try patching different SQL constructs together to produce novel SQL statements. Since the answers to problems in this domain are SQL statements, we can use this technique to craft new *problems*.

The basis of our work was therefore to produce a representation for CBM that allows it to be used for problem solving, and to exploit this capability by adding

algorithms for solving student problems and for generating new exercises dynamically.

1.5 Thesis contributions and outline

Constraint-based modelling is a promising approach with a plausible psychological foundation. However, experiences with SQL-Tutor suggest that while such models are fairly easy to build, on their own they are of fairly limited utility. This thesis explores the practicalities of building ITSs using CBM. It proposes a representation for constraints, and a set of algorithms that extend the capabilities of CBM to problem solving and problem generation. It proposes and experimentally evaluates the following four hypotheses:

- **Hypothesis 1**: It is possible to build a constraint-based domain model that contains sufficient information to solve problems and correct student solutions, by adopting a constraint representation that makes all of the logic in each constraint reversible;
- **Hypothesis 2**: Using the representation defined in hypothesis 1, it is possible to develop an algorithm for solving problems and correcting student answers, which does not need further domain information to achieve this;
- **Hypothesis 3**: CBM can also be used to generate new problems that fit the student's current beliefs, and this is superior to selecting one from a pre-defined list;
- **Hypothesis 4**: Because the new representation is domain-independent, it may form the basis of an ITS authoring tool that supports the development of new CBM tutors.

For hypothesis 1 to be true for a given domain, the constraint representation must be sufficiently expressive that it can describe the entire model without relying on external functions, yet simple enough that all operations it performs (such as testing for a valid value of a term) can be *reversed*, i.e. given that term *t* is valid, we can say

why it is so. ITS domains come in many different types, such as procedural, declarative and open-ended. Whether hypothesis 1 is true for all domains (and if not, what characterises the domains for which it *is* true) remains an open question. In this thesis we explore two domains: the SQL database query language, and English vocabulary and spelling. We develop a representation that is suitable for both these domains.

Similarly, the algorithm we develop to demonstrate hypothesis 2 works quite well for the SQL domain but is not guaranteed to work for all others. Also, the behaviour of the algorithm relies heavily on the completeness and correctness of the domain model. In Chapter 5 we demonstrate that the algorithm performs satisfactorily in the SQL domain but is not flawless, because of problems with the constraint set. Instead of trying to prove hypothesis 2 for all domains, we set ourselves the practical target of showing that hypothesis 2 is *feasible* in that solution generation can be performed acceptably in a complex domain such as SQL, despite errors in the domain model.

For hypothesis 3 we discuss the possibility of generating problems on the fly in Chapter 6, but we do not demonstrate that it works. Again, we are at the mercy of the constraint set, which makes the approach risky. Instead, we propose a more practical solution: we use problem generation to create a large problem set offline, which increases the chance that the system will choose a suitable problem in a given situation. In doing so, we develop a novel method of determining problem difficulty based on the constraints, which is needed to create appropriate problems on the fly. This new difficulty measure turns out to provide a more accurate means of problem selection.

We have built an ITS authoring tool that uses our new representation. The domain model is entirely represented in data files using the new constraint language, with no added code for external functions. We therefore satisfy hypothesis 4 for the representation chosen. Again, it remains an open question whether there are types of domains for which this would not be possible.

An outline of the thesis structure follows. In Chapter 2, we briefly describe the fundamentals of ITS, and give details on the current state of the art, Cognitive tutors. We then introduce constraint-based modelling, and discuss how it compares to

Cognitive tutors, and show why CBM is a worthwhile approach to research. We also describe SQL-Tutor, an example of CBM applied to a complex domain.

Chapter 3 describes the limitations of CBM as implemented thus far and gives the motivation for our work. In Chapter 4 we introduce a new representation for constraints that we have developed, which is designed to be easy to use and readily reasoned about by the system. This representation forms the basis of the work in the next three chapters. Chapter 5 discusses the idea of solving problems (and correcting student answers) directly from constraints, and details the algorithm we have developed. This algorithm makes it possible for the system to return (as feedback) a corrected version of a student's incorrect answer. We also give the results of a laboratory evaluation of this approach. In Chapter 6 we extend the approach to generating novel SQL statements, and describe how this is used to build new problems for the student to solve, based on their current student model. We present the results of both a laboratory test and a six-week classroom evaluation. We also describe a method for inducing high-level student models using machine learning, which we developed while trying to determine the best way to select target constraints for problem selection.

The purpose of our research is to facilitate the authoring of new CBM tutors. In Chapter 7 we describe an authoring system we have implemented for building text-based CBM tutors. We have reimplemented a tutor (SQL-Tutor) using this system, and built a new system for teaching English language skills. Both are described. We also discuss an algorithm for building new CBM domain models based on the MARVIN machine learning system. Finally, we summarise the results of our research and reiterate fruitful areas for future work. This thesis makes the following contributions to research in ITS:

- We develop a representation for constraints that is simple and transparent, and show that it is sufficiently expressive for two domains—English vocabulary and SQL—the latter being structurally complex;
- We present an algorithm for correcting student solutions that uses only the constraints to guide it, and allows constraint-based tutors to show the student

hints about what they should have done. We demonstrate its feasibility in a complex domain (SQL);

- We develop an algorithm for generating new problems from the constraint set and demonstrate its efficacy in the SQL domain. We also introduce a method for determining the difficulty of each problem with respect to an individual student, and demonstrate via a classroom evaluation that a system using both problem generation and the new difficulty measure outperforms one that uses neither;
- We implement a constraint-based authoring system that uses the new representation and demonstrate its effectiveness in the domains of SQL and English vocabulary. We show through a classroom evaluation how the latter is an effective tutoring system, despite being built in a very short time by someone who was not an expert in teaching that domain;
- Through all the above, we increase the practicability of implementing constraint-based tutors, and thus make a significant contribution to the field of ITS.

In the course of this research, we have prepared and presented 11 publications, which are listed in Appendix C.

2 Background

2.1 Intelligent tutoring systems

Computers have been used in education since the sixties (O'Shea and Self 1983). The first Computer Aided Instruction (CAI) systems presented material to the student in a static “storyboard” fashion, where every student received the same material, although they may have had some control over how they navigated through the curriculum. At appropriate (again, static) intervals, the system posed questions for the student to answer. The earliest CAI systems (so-called “linear” systems) assumed that the student’s answer would nearly always be correct, and that the system needs modification if this is not true (O'Shea and Self 1983). Later systems included “branching”, where the response to a student’s answer differed according to what their response was. However, because these CAI systems lacked any knowledge of the domain being taught, specific feedback was difficult, because it had to be hand-crafted for each problem. As a consequence, the system’s response was often limited to indications of right/wrong or presentation of the correct answer, and so the problems posed usually required only yes/no, multi-choice or a short (e.g. numeric) answers.

CAI systems can achieve modest gains in learning performance over classroom learning (Kulik, Kulik and Cohen 1980), however this falls short of individual one-on-one (human) tutoring, which may improve students' learning performance by up to two standard deviations (Bloom 1984). This prompted researchers to investigate ways that computer-based teaching environments could more closely approximate human

tutors. Many approaches have been tried, some very anthropomorphic, such as animated agents (Johnson, Rickel and Lester 2000) and natural language dialogue systems (Petrie-Brown 1989). The latter allows computers to emulate classic tutoring behaviour such as Socratic dialogues.

Intelligent Tutoring Systems (ITS) may mean any system that uses advanced techniques such as those described to improve teaching/learning performance. However, in more recent times ITS has come to mean teaching systems that "care" (Self 1999). Self describes "care" as meaning that ITSs are sensitive to what the student knows, what they misunderstand, and what they want to do. In other words, ITS attempts to tailor the system to the individual using it.

Even under this more restrictive definition of ITS there remain many different possible approaches. Cognitive Tutors (Anderson, Corbett, Koedinger and Pelletier 1995) provide a problem-solving environment with rich feedback. Collaborative learning systems (Dillenbourg and Self 1992; Soller, Goodman, Linton and Gaimari 1998) try to facilitate positive interaction between students by promoting interaction, encouraging participation, supporting collaborative skill practice and promoting group processing, rather than directly tutoring each individual knowledge in the domain being learned. Computer coaches such as SHERLOCK (Lajoie and Lesgold 1992) present the system as both an environment in which the student can practise tasks, and as a more advanced peer who can lead the student through impasses and thus enable them to work on problems that would otherwise be out of reach. Simulation tutors (Alexe and Gescei 1996; Satava 1996; Yacef and Alem 1996; Forbus 1997; Munro, Johnson, Pizzini, Surmon, Towne and Wogulis 1997; Rickel and Johnson 1997) provide an environment in which the student can experiment in the chosen domain with computer direction. Some tutors fall into more than one of these categories: for example, SHERLOCK is both a simulation tutor and a coach.

In this thesis we are primarily interested in systems like the Cognitive Tutors, which support learning by problem solving. The student is given a problem in the chosen domain, which they attempt to answer. The main roles of the system are to set the problem, and to provide rich help and feedback as the student progresses. Lajoie (Lajoie 1993) identifies four types of cognitive tools that can be identified by the functions they served: those that (1) support cognitive processes such as memory and

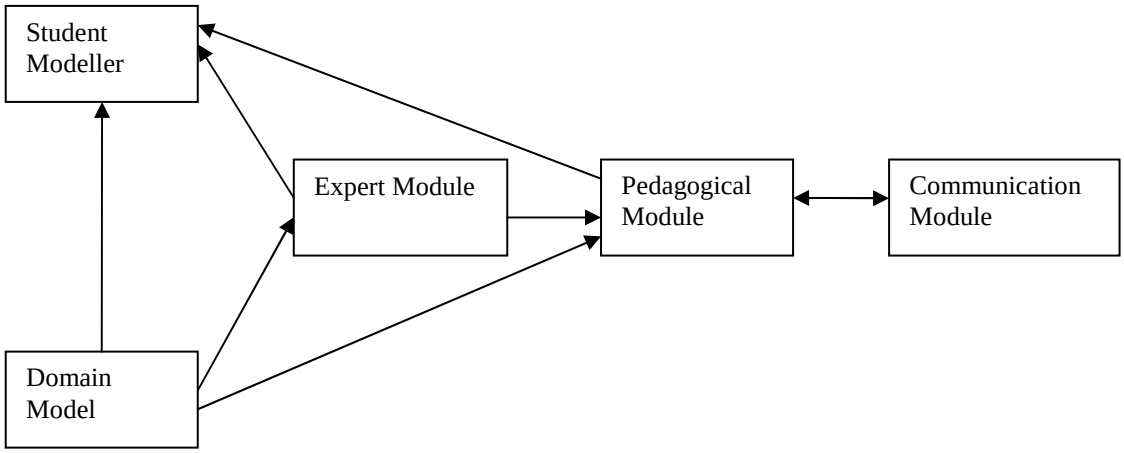

Figure 1. Architecture of an ITS

metacognitive processes, (2) share the cognitive load by providing support for low-level skills, (3) allow learners to engage in activities that would otherwise be beyond their reach, and (4) allow learners to generate and test hypotheses in the context of problem solving. The ITSs we are concerned with cover at least the last three of these. However, they most strongly fit category 3. By providing rich and detailed feedback during problem solving, they allow the student to tackle problems that they would be unable to solve on their own. In this context they are like an individual human tutor coaching a student through a difficult problem by teaching them the knowledge and skills they currently lack to complete the exercise.

2.1.1 Architecture

Many different architectures exist for intelligent tutoring systems. However, most share a common set of functional units as shown in Figure 1 (Beck, Stern and Haugsjaa 1996). Each of these is now described.

2.1.2 Domain model and expert module

The domain model contains a representation of the information to be taught. It provides input into the expert module, and ultimately is used to produce detailed feedback, guide problem selection/generation, and as a basis for the student model.

The domain model may take many forms, depending on the knowledge representation used, the domain it represents, and the granularity of the information

being represented. In page-based systems such as adaptive hypertext (Brusilovsky 2000) or adaptive storybook systems such as those produced by the REDEEM authoring system (Ainsworth, Grimshaw and Underwood 1999), domain knowledge is stored at the page level, and provides basic information about the content of the page, which aids in problem selection and course sequencing. In Cognitive tutors, the domain model consists of low-level production rules that completely describe the expected student behaviour down to "atomic" thought components (Anderson and Lebiere 1998). Simulation-based systems, e.g. RIDES (Munro, Johnson, Pizzini, Surmon, Towne and Wogulis 1997), use the domain model to describe how each component of the simulation should behave (i.e. what actions are possible with this object, and what the consequences of each action should be), and how components are interrelated. Constraint-based systems describe the possible (and pedagogically interesting) valid states that an answer may occupy.

The expert model uses the domain knowledge to advise other parts of the system. It may indicate the relative difficulty of curriculum sections or problems, such that the pedagogical module can select the next task. In Cognitive tutors it identifies whether or not the student's current solution is on track and, if not, what has gone wrong. It may also be able to run the domain model to solve the problem from a given state. In constraint-based systems it evaluates the student solution against the constraints to determine what concepts have been misapplied.

2.1.3 The Student model

The student model contains information specific to each individual student (or, possibly, populations of students), which is used to tailor the system's response to individual needs. The student model does not actually *do* anything by itself. Rather, it provides input to the pedagogical module.

Because student modelling is so central to ITS, it is also a controversial area. Initially, the goal was for the student model to model the student's mental state as completely as possible. Modellers therefore tried to represent many different mental constructs and attributes, such as learned facts and omissions in knowledge, mal-formed knowledge, relevant real-world experiences, attentiveness, tiredness, and so on. The task quickly became impossible and pessimism set in. Then, in 1988 Self

published the paper “Bypassing the intractable problem of student modelling”, which sought to find a solution to the impasse (Self 1990). Self proposed four “rules” of student modelling, which sought to overcome the pessimism and silence some of ITSs detractors. They are:

1. **Avoid guessing.** Have the student supply information such as the current goal, if it is needed by the system, rather than trying to infer what they are doing. This decreases the requirements of the system, and reduces the likelihood of making decisions about actions based on incorrect assumptions;
2. **Don’t diagnose what you can’t treat.** There is no point in modelling information that will never be used by the pedagogical module. Rather than trying to model everything you can about the student, decide what pedagogical actions you wish to take, and build the student model to support it;
3. **Empathise with the student’s beliefs.** Don’t label them as bugs if they disagree with the system, but rather strive to converge the system and the user beliefs. This means, for example, being mindful that the student might be correctly solving the problem, but in a different way to the system. Strive for sufficient flexibility that the system can accept, and adapt to, different problem-solving approaches;
4. **Don’t feign omniscience.** Assume the role of “fallible collaborator”. That is, allow the model to be overridden by the student, rather than taking complete control and refusing to relinquish it.

Following Self’s paper, there has been much more research into student modelling, with many different systems being devised. However, most of these fall into three main approaches (Holt, Dubs, Jones and Greer 1994): overlay models, perturbation models, and other methods. Each is now described.

Overlay models

These assume that the domain model contains all of the concepts the student must learn, and that the student knows a subset of this information. The task of teaching is therefore seen as filling in the holes in the student’s knowledge until they have learned sufficient of the model to achieve mastery. For a production rule domain

model this means that all rules have been applied with sufficiently few errors that they can be said to be learned. In an adaptive hypertext system all of the curriculum has been either covered, or is considered learned (because more comprehensive material has been covered), with enough problems on each page answered correctly. In a simulation the student may have successfully applied all the procedures they are required to learn (such as resuscitating the simulated drowning victim). A variation on overlay models is the *differential* model, which assumes that different parts of the domain have different importance, and so models the difference between student knowledge and the *expected* student knowledge rather than the entire domain. The expected subset may vary over time, thus "forgiving" some gaps in the student's knowledge early on but remediating them later, as the expected model changes to require them. WEST (Burton and Brown 1978), a gaming system for teaching arithmetic, is an example of such a differential model.

Whilst many different systems fall into the general category of overlay systems, they may vary greatly in their specific implementations, particularly how they judge which parts of the domain are learned and which are not. The simplest method is to consider a knowledge element learned after it has been successfully applied *n* times. Some Cognitive Tutors (Anderson, Corbett, Koedinger and Pelletier 1995) use a complex Bayesian formula to calculate the probability that each production rule has been learned. It takes into account several *a priori* probabilities: that the student already knew the production, that they will learn it at a given opportunity to apply it; that they will correctly guess how to use it, and that they will accidentally misuse it, even if they know it. Then, after each step, they calculate and update the probability that the student actually knows the rule, given their observed performance. This information can then be used to predict performance, and thus help select new problems. Mayo and Mitrovic (Mayo and Mitrovic 2000) use a similar method for a constraint-based student model. The Cardiac Resuscitation simulation system, or Cardiac Tutor (Eliot and Woolf 1995), estimates the desirability that the system end up in a given simulation state (e.g. patient fibrillating) given the student's behaviour so far, and the probability that the student's behaviour will lead it to this state. This information is used to adjust parameters of the simulation such that the desired state is more likely to be reached. Many other schemes exist.

Perturbation models

Whereas an overlay model assumes that the student's knowledge is a subset of that of an expert, a perturbation model recognises that the student may harbour misconceptions, or "buggy knowledge", which must also be represented. Figure 2 illustrates the notion of a perturbation model.

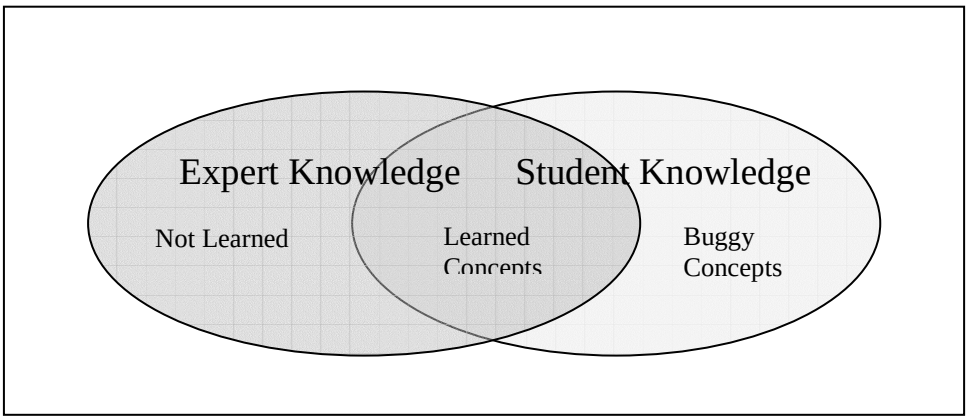

Figure 2. Perturbation model

Building a perturbation model usually requires that the underlying domain model contain information about the mistakes students are likely to make, or "bug libraries" (Burton 1982) so that they can be identified in individual students' behaviour. For example, model-tracing tutors often include incorrect productions, which represent commonly made mistakes.

Other approaches

Researchers have recently begun to use machine learning to try to induce student models. ADVISOR (Beck and Woolf 2000) uses a functional approximator to predict the time taken to solve the next problem and the probability that the student will answer it correctly, based on the complexity of the problem, the student's proficiency, and the number of hints they have received for this problem. In Section 6.2 we use a variation of the rule induction algorithm PRISM (Cendrowska 1988) to infer a high-level student model from the low-level constraint information (Martin 1999). Gilmore and Self (Gilmore and Self 1988) similarly use an ID3 type classification system to learn the concepts the student has and hasn't learned.

2.1.4 Pedagogical module

This module decides what to present to the student next. It uses information from the student, domain and expert models to arrive at each decision. In effect, this module models the "teaching style" to be applied. For example, it may favour examples over the presentation of static text. It may make both low-level decisions, such as the level of difficulty of practice exercises, and high-level ones, such as when the student should move to the next topic of the curriculum. There are many different forms, depending on the teaching style being modelled, and the kind of information available: adaptive hypertext systems may only control page presentation, while the pedagogical module of a Cognitive tutor may determine problem difficulty, level of feedback, and when to declare that a portion of the curriculum is learned.

2.1.5 Communication Module

Also known as the interface module, it interacts with the learner, displaying information and accepting input from the student.

In this research, we are primarily interested in the domain and student models. With the exception of the authoring system in Chapter 7, which is a complete system built around CBM, we have intended the methods we have developed to be independent of interface or pedagogy.

2.2 The state of the art: Cognitive Tutors

Cognitive tutors (Anderson, Corbett, Koedinger and Pelletier 1995) model the domain to be learned as a runnable model (such as a set of production rules), that map out all the valid ways a student may solve the problem. Based on Anderson's ACT-R theory or "rules of the mind" (Anderson 1993; Anderson and Lebiere 1998), they were initially proposed partially as a means of validating Anderson's theories. Since then they have become successful tutors in their own right, the most celebrated being the Algebra Tutor, which has been shown to provide gains of 1 SD (over non-tutor users) in the subject of secondary school algebra (Koedinger, Anderson, Hadley and Mark 1997).

2.2.1 ACT theory: rules of the mind

The central tenet of Anderson's theory is that the processes of thought can be modelled using declarative and procedural knowledge. Whereas others have argued over which of these (if either) are a better representation of the mind, e.g. (Winograd 1975), Anderson contends that both are necessary. To be able to perform a task, a person needs the required procedural knowledge. However, before they can learn this they first need the underlying declarative knowledge. Learning then becomes a two-step process: first the student must acquire the appropriate declarative knowledge, and then they must develop this into the required procedural knowledge. Although humans can perform tasks where they have forgotten the declarative knowledge that led to their acquisition of the skill, Anderson asserts that they must have had that knowledge *at some point*.

Anderson produced a formal representation for declarative knowledge using *chunks* (Miller 1956) and procedural knowledge (production rules) to describe a person's knowledge state. This representation, plus the rules for their use and creation, forms the ACT theory. Since its inception in 1976, the ACT theory has been modified heavily, giving rise to several major versions: ACTE, ACT*, and several versions of ACT-R, of which the current is ACT-R 5.0. During this time Anderson has refined the definitions for chunks and procedural rules by restricting what can be represented in a single chunk or rule, and the way in which they can be generated. For example, a production rule may only produce one of the following six combinations of effects: no change, goal elaboration (current goal modification, no change to the goal stack), side effect (new goal on stack, no modification to current goal), return result (modify current goal, push new goal on stack), pop unchanged (pop completed goal without modification), and pop changed (modify current goal and pop off the stack). He believes his latest representation models the atomic components of thought. This is backed by empirical evidence: given an appropriate latency for the firing of each production rule and the retrieval of each chunk, simulations of tasks using Anderson's model display behaviour where the time taken to perform a task correlates closely with human performance on a wide variety of tasks.

2.2.2 ACT-R and learning

Any theory that describes the structures of thought must also be able to describe how they got there. ACT-R contains theories of learning for both declarative and procedural knowledge. As stated previously, production rules can only make modifications to goals. Since productions are also the only means of performing a mental action, goal chunks must be used to store new declarative knowledge. In ACT-R 4.0 when a novel problem is solved, a goal chunk is stored that essentially represents the solution to the solved problem (such as “the answer to 6+4 is 10”). If a person later solves a problem that is similar to an existing one, the two may be merged to form a more general declarative chunk. Thus declarative knowledge may be obtained from performing some procedure and remembering the result. For example, some children learn their addition tables by using counting to add numbers and remembering the answer. Note that this learning process is not deterministic: having noted that 6 plus 4 is 10, the child may subsequently fail to remember this fact and again resort to counting. However, the more times the child encounters the problem 6+4, the more likely they are to remember the answer. This is born out by experiment.

ACT-R 4.0 also describes the learning of procedures. As a person performs some procedure, they may at intervals seek to understand what they have done, so as to be able to repeat the task. ACT-R represents this via a *dependency* goal, which specifies the relationship between two (encountered) goals and the constraints upon when this dependency is valid. Consider the following point in a multi-column addition problem, which a child is being shown how to perform:

```
 23
+34
---
 7
```

At this stage the student is aware that 3 + 4 = 7. They have an initial state where no numbers have been filled in, followed by the next state where “7” has been written. They also have a declarative chunk that corresponds to the two numbers in the

column, namely "7 is the sum of 3 + 4". They now create the dependency goal (in pseudocode):

```
Initial goal is ADD-COLUMN with values 3 and 4, and an answer of
NIL
Modified goal is ADD-COLUMN with values 3 and 4, and an answer of 7
Constraint is FACT34: ADDITION-FACT 3 + 4 = 7
```

ACT-R now pops this dependency goal and creates a procedural rule from it. Since the values "3" and "4" appear in the initial and modified goals, plus the dependency, ACT-R assumes that such repetition is not coincidence, but that it indicates that such terms can be variablised, so the following rule can be induced:

```
IF
   Initial goal is ADD-COLUMN with values N1 and N2, and answer NIL
AND
   There exists ADDITION-FACT N1 + N2 = SUM
THEN
   Modify goal to ADD-COLUMN with values N1 and N2, and answer SUM.
```

ACT-R 4.0 also provides a theory for the learning of the sub symbol parameters, i.e. how fast each piece of declarative knowledge can be retrieved and the speed and utility of performing the production rules. Together these four aspects of learning (creating declarative chunks, merging/generalising chunks, dependency goals for procedural learning, and retrieval speed and utility) constitute a robust theory of learning based on ACT-R's definitions of the atomic components of thought, which are well corroborated in practise.

2.2.3 Cognitive tutors

Cognitive tutors were initially developed in part to validate the (then) ACT* theory of mind. An early goal was that the tutors should possess a plausible model of what the student was trying to learn:

"*The core commitment at every stage of the work and in all applications is that instruction should be designed with reference to a cognitive model of the competence that the student is being asked to learn. This means that the system possesses a computational model capable of solving the problems that are given to students in the ways students are expected to solve the problems.*"

(Anderson, Corbett, Koedinger and Pelletier 1995).

Thus, from the outset Cognitive tutors have followed (versions of) the ACT-R theory of learning as described above, which continues to this day. Recall that learning in ACT-R is described as the acquisition of declarative knowledge chunks, followed by the compilation of procedural rules that apply declarative knowledge to tasks being mastered. Anderson believes that the acquisition of declarative knowledge is relatively straightforward and unproblematic compared with the subsequent refinement into procedural knowledge. Hence, Cognitive tutors focus on the acquisition of production rules and represent their domain models in this manner, supported by declarative knowledge chunks, which are assumed to already be learned.

Initial work on Cognitive tutors was intended to support the ACT theory of skill-acquisition by showing that learning could be achieved by getting students to behave like the production-rule model. This required from the outset that the domain model be a complete model of the task being performed, specified using production rules. Tutoring is achieved using a method known as *model tracing*. As the student works at the problem, the system traces her progress along valid paths in the model. If she makes a recognisable off-path action she is given an error message indicating what she has done wrong, or perhaps an indication of what she should do. If the action is identified as being off-path but cannot be recognised, she may only be told that it is incorrect, but not why. Because of the combinatorial infeasibility of tracking the student's state relative to the correct path throughout the entire domain space, early Cognitive tutors forced the student to stay on a correct path. Later tutors relaxed this requirement somewhat, however this was done in an *ad hoc* way. For example, later versions of the LISP tutor (Anderson, Farrell and Sauers 1984) allowed delayed feedback, which dropped the necessity for the student to stay on a recognised path. Instead, if the student produced a program that could not be recognised, it was run against a set of test cases and accepted if it returned the correct results. This has the disadvantages that specific feedback may not be given for an incorrect solution, and that the student might "get lucky", producing an incorrect program that happens to work for the test cases.

Analogous to the model tracing technique for critiquing the student's action is the student modelling method of knowledge tracing. A Bayesian procedure is used to estimate the probability that a production rule has been learned after each attempt. Formulas 1 and 2 give the probability that a production is in a learned state after a correct answer and an error, respectively.

$$p(L_{n-1} \mid C_n) = \frac{p(L_{n-1}) * (1 - p(S))}{p(L_{n-1}) * (1 - p(S)) + p(U_{n-1}) * p(G)} \quad (1)$$

$$p(L_{n-1} \mid E_n) = \frac{p(L_{n-1}) * p(S)}{p(L_{n-1}) * p(S) + p(U_{n-1}) * (1 - p(G))} \quad (2)$$

$p(L_{n-1})$ is the probability that the rule was already learned, $p(S)$ is the probability of a slip, $p(U_{n-1})$ is the probability that the production was *not* previously known, and $p(G)$ is the probability that the student guessed.

This information is used to decide when the skills of a section of curriculum have been satisfactorily learned. Anderson et al. found that these skill probabilities could be used to accurately predict post-test performance (Corbett and Anderson 1992).

2.2.4 Example: LISP TUTOR

The LISP tutor (Anderson, Farrell and Sauers 1984) was an early attempt at a Cognitive tutor. The student is given a description of a small program to encode in LISP, which she then writes with the system's help. Interaction is similar to using a (very comprehensive) language-sensitive editor. As the user builds their solution, the system inserts tags that describe the general form of the program, for example (user input in bold):

```
(defun fact (n)
   (cond ((equalp) <ACTION>)
     <RECURSIVE-CASE>))
```

The standard LISP tutor does not allow the student to stray at all from the model of desired performance. In the above example, the student needs to test for zero, for which there is a dedicated function. The tutor immediately interrupts the student and

makes them use the ZEROP function, rather than the more general EQUALP. In fact, the use of EQUALP is valid, however this would lead to a solution that is off the path specified in the LISP tutor, because the authors have deemed it undesirable. Note that this is a design decision, rather than a characteristic of the modelling method. Further, the LISP tutor will interrupt the student at key points, even if no recognisable mistake has been made, if they perform actions that are not expected. In the previous example, this student is writing a recursive function. Anderson et al. determined that students tend to find the terminating case easy to write, but struggle with the recursive case. Therefore, if they perform any unexpected actions after writing the terminating case, the system asks them a question about the recursive case, even though they may be performing some valid action. If they fail to answer the question correctly, the system digresses with some exercises to help them understand the nature of the recursion they are trying to build. Again, this is an issue with the model being used, but it highlights how deficiencies in the model may lead to overly prescriptive behaviour.

In order to follow the student correctly, the LISP tutor needs to constantly know their intent. In many cases this is obvious, but in others (such as declaring one of several variables) further clarification is needed. The LISP tutor pops up a menu whenever something needs to be disambiguated. Once they have finished the exercise, they are presented with a standard LISP environment in which they can test their code.

The LISP tutor performs very well. In an initial mini-course at CMU, students using it solved a series of exercises in 30% less time than those in a standard LISP environment, and performed one standard deviation better on their final test. As a result, a full year course was devised using the LISP tutor, which continues to this day. However, later evaluations failed to conclusively prove that the style of teaching used (notably comprehensive, immediate help) improved performance *per se*. Anderson et al. concluded that the main reason for improved performance in post-tests previously was probably because the LISP tutor enabled students to cover more exercises in the same amount of time, which subsequently gave them an advantage. However, this in itself is considered a worthwhile outcome, since enabling students to achieve their learning in less time gives them more time to learn additional material, or to do other things.

2.2.5 Summary

Cognitive tutors are some of the most successful to date. The PAT tutor for high school mathematics has produced gains as high as 100% for areas targeted by the system (Koedinger, Anderson, Hadley and Mark 1997). The model-tracing technique, on which they are based, is derived from a comprehensive theory of learning, ACT-R, and the results obtained with it strongly support that theory.

2.3 Motivation for change

Since Cognitive tutors are so effective and the technology for building them is well understood, it may seem unnecessary to explore other options. However, they harbour several outstanding issues. In particular, the following may be attributed directly to the Cognitive tutoring method, rather than to the wider field of ITS in general (Anderson, Corbett, Koedinger and Pelletier 1995): Cognitive tutors are hard to build, they may be too restrictive, and they may not suit all domains. These three issues are all related.

With respect to difficulty in building tutors, Anderson estimates the typical time to author a system is around 10 hours per production (Koedinger, Anderson, Hadley and Mark 1997), although this figure is not backed up by empirical data, and seems overly large. For example, Koedinger authored the 25 productions for Kermit in just a few days (See section 2.4.3). Complex domains may run into hundreds, perhaps thousands, of productions. Using the 10 hours estimate, a very simple tutor for teaching subtraction with carry requires at least six production rules (Blessing 1997), so would require more than a week's effort, and a domain such as SQL might take years just to author the production rules. Such effort would be a serious barrier to building tutors for very complex domains, yet this is where the need is arguably greatest.

Some domains are highly suited to a procedural approach. In arithmetic for example, there tend to be a few well-defined procedures for performing tasks such as addition, subtraction, and division. In others such as programming, this is not the case. Commenting on tutoring systems for programming in general, (Deek and McHugh 1998) note that:

> *"Intelligent systems present the student with a simple problem containing a clear definition, specifications and constraints. The student is then **led** into finding the 'ideal solution'."*

A consequence, they claim, is that students become *dependent upon* being led to the solution, and fail to develop the skills to determine the solution for themselves. Part of the reason for this is that often only a single solution path is encoded in the production rules. In the LISP tutor for example, the student is guided very closely along a particular path. For example, alternative means of testing a condition may be discounted by the tutor. Allowing greater flexibility in programming language tutors is difficult because they often contain a high level of redundancy. For example, in SQL there are three completely different *problem-solving strategies* for retrieving data from multiple tables. The following three queries all perform the exact same function (retrieve the name of the director of "Of mice and men"), but use different strategies:

```
SELECT lname, fname
FROM movie, director
WHERE director = director.number and title = 'Of mice and men'

SELECT lname, fname
FROM movie join director on movie.director = director.number
WHERE title = 'Of mice and men'
SELECT lname, fname
FROM director
WHERE number =
      (select director from movie where title = 'Of mice and men')
```

There is no obvious "ideal" solution to the above problem, although there may be criteria with which one could judge merit (e.g. efficiency). Further, there are many subtle details that could be modified arguably without affecting the quality of the solution, such as whether or not to qualify names (e.g. "director.lname"), and whether or not to use table aliases to shorten name qualifications. While such alternatives could be represented using the production rule approach, it would be a substantial undertaking. Even successful Cognitive tutors such as the LISP tutor are sometimes criticised for being too restrictive because they inevitably exclude valid solutions (Anderson, Corbett, Koedinger and Pelletier 1995; Deek and McHugh 1998), although it is arguable whether or not this affects learning. Since Cognitive Tutors

have been shown to be capable of producing dramatic learning outcomes (Koedinger, Anderson, Hadley and Mark 1997), this might even be a positive feature.

More importantly, in SQL the solution is structured into six clauses representing different aspects of the problem: what to select (SELECT), where from (FROM), any restrictions (WHERE), grouping (GROUP-BY), sorting (ORDER-BY) and group restrictions (HAVING). There is no right or wrong way to approach writing an SQL query. For example, some students may choose to focus on the "what" part of the problem first, and then fill in the restrictions, whereas others may first attend to the restrictions, or even sorting. Again, it is possible to encode all variations as separate paths through the production rule model, but this would make the mode large and unwieldy. Worse, the actual paths represented are of *no importance to us*. The production rule model is simply too low-level an abstraction for this type of domain. Similarly in data (entity-relationship) modelling, it is equally valid to define all entities and then their relationships, or define each pair of entities and their relationships simultaneously. Thus it appears that there are domains for which Cognitive tutoring is likely to be an unwieldy (or possibly unworkable) tool for modelling.

What happens when there is no "right" answer to a problem at all? For example, imagine a tutor for musical improvisation, where the student's input is via an instrument such as a keyboard and the "problem" is an accompaniment that the student must improvise over. There is clearly no such thing as a "correct" answer to this problem. The domain is procedural in that the student is performing the procedure of playing a note, followed by another one, where each action (note) will have many characteristics, such as pitch, volume, timing and colouration (bending, tremolo, slide, etc), however there is no correct procedure to follow other than "after playing a note, either play another one (sometime) or finish". It is not at all obvious how a production rule model could be built for such a domain, and yet there are still many ways an ITS could provide useful feedback, such as how well the notes played fitted the key of the piece, whether the student's playing was an example of the style (e.g. "blues") being practised, and how "interesting" the piece was (as judged by the tutor's author).

In summary, although Cognitive tutoring has been shown to work extremely well for some domains, there are others for which they may be less suitable, or even

impossible to implement, although we have not explored this in any detail. In particular, open-ended domains such as the musical improvisation tutor described seem less suited to the Cognitive Tutor approach, because they do not require a model of procedure, and might not benefit from the effort of building one. This might also apply to more declarative domains such as SQL. There is therefore scope for alternative methods that can cope with, and are suited to, declarative and open-ended domains. The research presented in this thesis is concerned with one such alternative, constraint-based modelling.

2.4 Constraint-based modelling

In 1994, Ohlsson proposed another method for domain and student modelling that is also based on a psychological theory of learning. Both the underlying theory and the resulting modelling system are radically different from Anderson's, and represent a major change in direction. The theory and model are now described. We also compare CBM to Cognitive tutors, and describe an existing system that uses CBM, to illustrate how it is implemented.

2.4.1 Learning from performance errors

As described in Section 2.2.2, Anderson's ACT-R theory included a theory for how new knowledge is learned by the creation of new declarative chunks and the compilation of new production rules. The former, he asserts, happens automatically via the retention of new problems and their solutions. The latter, according to his later theories in ACT-R 4.0, occurs when the student makes a conscious decision to reflect on how they just performed some step of the solution. Tutors based on ACT-R therefore concentrate on keeping the student on a valid solution path, such that they commit correct productions to memory, not buggy ones they have arrived at erroneously.

Ohlsson (Ohlsson 1996) has a different view. In a theory called "learning from performance errors", he asserts that procedural learning occurs primarily when we catch ourselves (or are caught by a third party) making mistakes. Further, he contends that we often make errors even though we know what we should do, because there are

simply too many things to think about, and we are unable to make the correct decision because we are overloaded. In other words, we may already have the necessary declarative knowledge, but for a given situation there are too many possibilities to consider for us to determine what currently applies. Thus merely learning the appropriate declarative knowledge is not enough: only when we have internalised that knowledge—and *how to apply it*—can we achieve mastery in the chosen domain.

We can represent the application of a piece of declarative knowledge to a current situation by describing the current problem-solving *state*. Ohlsson uses constraints for this task. Each constraint consists of a relevance and a satisfaction condition. The first specifies when this piece of declarative knowledge is relevant, and the second describes the state whereby this piece of knowledge has been correctly applied, i.e.:

```
IF <relevance condition> is true
THEN <satisfaction condition> will also be true
```

Consider a person learning to drive. On approaching an intersection, she must consider many factors regarding who gives way and decide whether or not to stop. Such pieces of knowledge relate, among other things, to the driving rules of the country she is in. In New Zealand for example, one such rule is that "at uncontrolled intersections, traffic on the right has right-of-way". Now, as our driver approaches an uncontrolled intersection, she must consider whether or not to give way. A constraint for the above situation might be (in pseudocode):

```
IF uncontrolled intersection
AND car approaching from right
THEN give way
```

By Ohlsson's theory, our learner driver may be flustered by the number of things she has to consider (especially if there are several other cars at the intersection), overlook the above constraint, and drive into the path of a car on her right. However, a skilled driver knows "intuitively" to look for the car on the right and stop because they have applied this constraint many times before, and it has been internalised in some way as procedural knowledge. The corresponding procedural rule in an ACT tutor might be:

```
IF the goal is to travel through the intersection
AND the intersection is uncontrolled
AND a car is approaching from the right
```

```
THEN set a sub goal to give way to the car
```

2.4.2 CBM in ITS

Whilst the underlying theories of ACT-R and Performance Errors may be fundamentally different, in terms of ITS implementation the key difference is the level of focus: ACT-R focuses in detail on the procedures carried out, while "learning from performance errors" is concerned only with *pedagogical states.* This translates directly into the domain models. Cognitive tutors faithfully model the procedures that should be learned, while constraint-based tutors represent just the states the student should satisfy, and ignore completely the path involved.

In a constraint-based tutor, the domain model is represented by a set of constraints, where each constraint represents a *pedagogically significant state*. That is, if a constraint is relevant to the student's answer, this is an example of a principle that we wish to teach the student. If the constraint is violated, the student does not know this concept and requires remedial action. A key test of whether or not a constraint represents a single pedagogically significant state (i.e. that all problems/solutions that fall into this state are pedagogically equivalent) is whether or not a single piece of feedback can be delivered for all problems that violate this constraint. Once the domain model has been so defined, we can associate feedback actions directly with the constraint. The basic definition of a constraint in a constraint-based tutor is therefore:

```
<constraint id>
<feedback action >
<relevance condition>
<satisfaction condition>
```

For example, in the domain of multi-column addition with carry, the following constraint (in pseudocode) checks that the student has correctly added the numbers in each column, where *ideal-solution* represents a correct solution to this problem, *problem* is the original problem specification, and *student-solution* is the student's attempt.

```
(1
```

```
"You have added two numbers incorrectly in column <N> – please
check your addition. Note that there is no carry for this column."

(and
          ideal-solution.column(N) = SUM
          problem.column(N).carry = NO
)

(student-solution.column(N) = SUM
)
```

The constraints are used to evaluate the student's input by comparing it to an "ideal solution". The ideal solution is just one of the set of possible solutions to the problem, and is considered "ideal" in the sense that it is the answer the author would ideally like the student to submit. However, it is *not* necessary for the student to submit this particular answer, nor to solve the problem in this particular way.

When the student submits a solution or action, each constraint is evaluated one at a time. Constraints may test elements of the student solution only (syntactic*)*, or compare aspects of the student and ideal solutions (semantic). For each constraint if the relevance condition is true, this constraint is relevant to the student's current solution or action. The satisfaction condition is then tried. If this is also true, the solution is correct with respect to this constraint and no further action is required. Otherwise, the constraint has been violated and the feedback action is taken.

The student model is also based on the constraints. The simplest is an overlay model, where the system determines that each constraint is either learned or not learned. There are various ways to classify each constraint. This is discussed further in Section 6.2.

2.4.3 Comparison with Cognitive tutors

The philosophies underpinning Cognitive Tutors and Constraint-Based modelling are fundamentally at odds. ACT-R asserts that learning is achieved when students reflect on a *successful* action, and internalise what they did as procedural knowledge. This requires a conscious effort on the student's part to assimilate what they have done. In contrast, Ohlsson asserts that learning occurs as the result of an *unsuccessful* action: to correct their mistake, students must apply their underlying knowledge to the current situation, and in doing so they automatically reinforce their internal knowledge of what to do. Ohlsson further believes that because the student is forced to reflect on the

declarative knowledge that underlies the action to be taken each time they are shown feedback, they are learning at a deeper level than simply remembering the procedure for solving a particular problem or kind of problem, and so their performance is more likely to be transferable to other problems and to the real world.

At the practical level, CBM has the following advantages (Ohlsson 1994):

1. The domain model is simple, and need not be runnable.

Cognitive tutors require a model of the desirable path from problem to solution so that they can trace the students' actions against it. This model must be complete and correct from the outset; otherwise the system cannot follow what they are doing. In contrast, a constraint-based model need only model pedagogically significant states, which in many cases is a much simpler task, because the number of factors that must be taken into account is less than the number of steps on alternative paths that a Cognitive tutor would need to model. Further, if a constraint is missing the effect is highly localised: the system simply fails to detect a particular type of error. Since constraints are modular, the rest of the solution should still be able to be assessed. This reduces the need to conduct large-scale empirical studies with a domain expert, and allows the domain model to be developed incrementally and deployed before the model is complete. For example, SQL-Tutor has been used for four years now, yet the model is still acknowledged to be incomplete. The initial version exhibited quite a few problems when used by a class, yet was still shown to be an effective teacher (Mitrovic and Ohlsson 1999).

To illustrate the difference in effort required, consider the domain of database (entity-relationship) modelling. KERMIT (Suraweera and Mitrovic 2001) is a constraint-based ITS in this domain. Consider the following two simple problem statements, which the student must represent by ER diagrams:

1. Some students live in student halls. Each hall has a unique name, and each student has a unique number.
2. Each student has a unique number, a first name, and a last name.

To assess answers to these two problems, KERMIT requires 23 constraints, such as (in pseudocode):

```
(1
```

```
"Each regular entity should have at least one candidate key
attribute"

(and
  (each student-solution.OBJECT),
  (OBJECT.type = entity)
)

(countof (OBJECT.ATTRIBUTE.type = key) >= 1)
)
```

In contrast, Koedinger implemented a procedural model for a Cognitive tutor that can assess the same two problems. It required 25 productions, 10 (trivial) general chunks and 30 problem-specific chunks, or a total of 55 major elements and 10 trivial ones. The 30 problem-specific chunks have no use outside these two problems, so are analogous to the ideal solution in the constraint-based solution, which consists of a list of *tags*, which represent the important features of the problem (entities, relationships, etc). For the example problems there are a total of 11 tags. Also, the production rules are typically more specific than constraints, so cover less of the domain. KERMIT has only 90 constraints in total, so in authoring the domain model for these two problems more than a quarter of the domain model has been implemented.

2. There is no need for a bug library

Cognitive tutors may optionally contain mal-productions as well as the correct ones. Without these, the model-tracer is unable to say *why* a step that is not on the correct path is wrong, and so is limited to a "that is incorrect" message, or demonstrating the correct next step.

In CBM, incorrect answers are implicitly encapsulated by the constraints: if a student has not added two numbers together correctly, they have implicitly made some error in their step. It may still be desirable to analyse students' answers to determine which parts of the domain are problematic, and so need to be modelled. It is also useful to observe student behaviour to help decide at what level student mistakes are pedagogically equivalent. However, the level of analysis required is less because it is not necessary to tag errors back to particular procedural steps.

3. There is no requirement for sophisticated inference mechanisms

Cognitive tutors need to know the intent of every student action, in order to decide what the current goal is, so that the action may be compared to the appropriate goal in

the model. Further, every action needs to be made available to the tutor. This requires either that the tutor perform evaluation after every student action (e.g., after every key stroke), or that the necessary details be inferred. In CBM, we are only interested in the *state* of the answer at any stage, rather than the sequence of actions or the intent of the student. We can therefore check constraints using simple pattern matching.

4. The model permits free exploration of the domain

A complaint of model-tracing tutors is that they are too restrictive (Self 1999). In particular, it is difficult to allow the student to perform explorative actions. Some systems do allow variation from the desirable path, but the extent to which this is practical is limited by the need to be able to determine when the student has gone completely off-track. The further the student is allowed to wander, the harder it is for the system to understand why they have deviated from the path and therefore to make judgments about whether or not they are completely lost. Worse, it quickly becomes almost impossible to make recommendations about how to get back on the path, apart from returning to the state where their solution first deviated, so the student may be forced to abandon a promising line of attack. SHERLOCK (Lajoie and Lesgold 1992) overcomes this by allowing multiple solution paths for each problem, but can still run into difficulty if the student keeps switching strategies. Many documented cases, e.g. (Ohlsson and Bee 1991), support the notion that "radical strategy variability" (i.e. complete changes of problem-solving approach) is the *normal case*.

CBM, on the other hand, is less troubled by strategy variation, since it does not try to track the student's problem-solving procedure. All that is therefore required is to *implicitly* represent all possible valid solutions. For example, in the SQL domain a constraint that tests that all tables are present must consider that tables may be represented in either the FROM or the WHERE clause, that the tables may be by themselves, in JOINS, in comma-separated lists, or in nested queries, and that table names may or may not be aliased. It need not consider whether the student is *trying* to implement a JOIN or nested SELECT. This obviates the need for multiple paths through the various options.

5. The model is neutral with respect to pedagogy.

Cognitive tutors need to follow every step the student takes and evaluate that step with respect to the correct solution path. For reasons of computational complexity, if the student strays from a solution path, it is necessary to get them back onto it quickly, before the task of determining the way back becomes computationally intractable. It is therefore necessary to remedy problems in a timely fashion, which dictates the teaching strategy used: evaluate every step, and provide immediate feedback if there is a problem. Note that this may be a deliberate decision: Anderson et al argue that the efficacy of this type of feedback is psychologically plausible, and they have conducted experiments with the LISP Tutor that show the immediate feedback leads to faster learning (Anderson, Corbett, Koedinger and Pelletier 1995).

CBM does not have this requirement. The solution may be evaluated at any time, since it is not necessary to be on any particular path. Partial solutions may be evaluated provided the system recognises that the solution is not complete, leaving tests for completeness until the student declares they are done. When the student submits her final solution, it is checked for completeness as well as correctness.

2.4.4 Applicability of CBM

The systems described in this thesis are all declarative in nature, in that the order in which student actions take place (i.e. the problem-solving *procedure*) are not considered relevant to the correctness of their behaviour. However, this does not mean that CBM can not be applied to procedural tasks. Consider the task of learning to count a set of objects. This domain requires a number of constraints upon the order of behaviour, such as (Ohlsson and Rees 1991):

- Always start with the first number in the numbering system being used (i.e. integers);
- Use the numbers in the order defined for the numbering system being used;
- Only use a new number if the one immediately preceding it has been used already;
- Do not count an object that has already been counted
- Do not cycle back to an object already counted

Ohlsson describes how CBM can deal with such rules by including into the student's current solution state *all of the actions taken so far*. Thus, if the solution state includes a set of the objects counted so far, a constraint may easily check that the current object being considered has not already been counted. Similarly, if the solution state includes an ordered list of the numbers used so far, it is easy to check this to ensure the current number is one greater than the last, and that the current number has not already been used.

Another example is the Cardiac Tutor (Eliot and Woolf 1995) . In this domain, the student is presented with a heart patient, which they must diagnose and treat. In this domain, both the sequence some actions and the timing of actions is important. CBM could model this by including, in both the problem and the solution, a trace of the required actions and their times. The constraints could then compare the order and timestamps of actions where it is important. In general, procedural domains may be handled by ensuring that the problem and solution states contain ordering and/or timing information.

In summary, CBM is a state-based approach that compares the state of the student and "ideal" solutions, to ensure that the student solution is always in a permissible state. It may be applied to any domain where the problem and solution can be presented this way. This may include procedural domains. However, for procedural domains with a relatively small level of branching in the trace of possible solutions, it might be more natural to use a model-tracing approach, since this immediately provides the benefit of being able to suggest the next action.

2.4.5 Example of a constraint-based system: SQL-Tutor

SQL-Tutor (Mitrovic 1998) is a CBM ITS that teaches the SQL database query language to Stage 2 and 3 students at the university of Canterbury. Several versions have been built, the latest being a web-enabled system, built using Allegro Common LISP, and the Allegroserve web server software (see www.Franz.com).

Figure 3 shows the user interface. In SQL-Tutor, constraints are encoded as LISP fragments, supported by domain-specific LISP functions. For example, in the following constraint, "attribute-in-db", "find-schema", "current-database" and "valid-table" are all specific to SQL.

```
; problem number
(p 147

; feedback message
"You have used some names in the WHERE clause that are not from
this database."

; relevance condition
(and (not (null (where ss)))
     (bind-all ?n (names
       (slot-value ss 'where)) bindings))

; satisfaction condition
(or (attribute-in-db (find-schema (current-database *student*)) ?n)
    (valid-table (find-schema (current-database *student*)) ?n))

; which SQL clause this constraint is most relevant to

"WHERE")
```

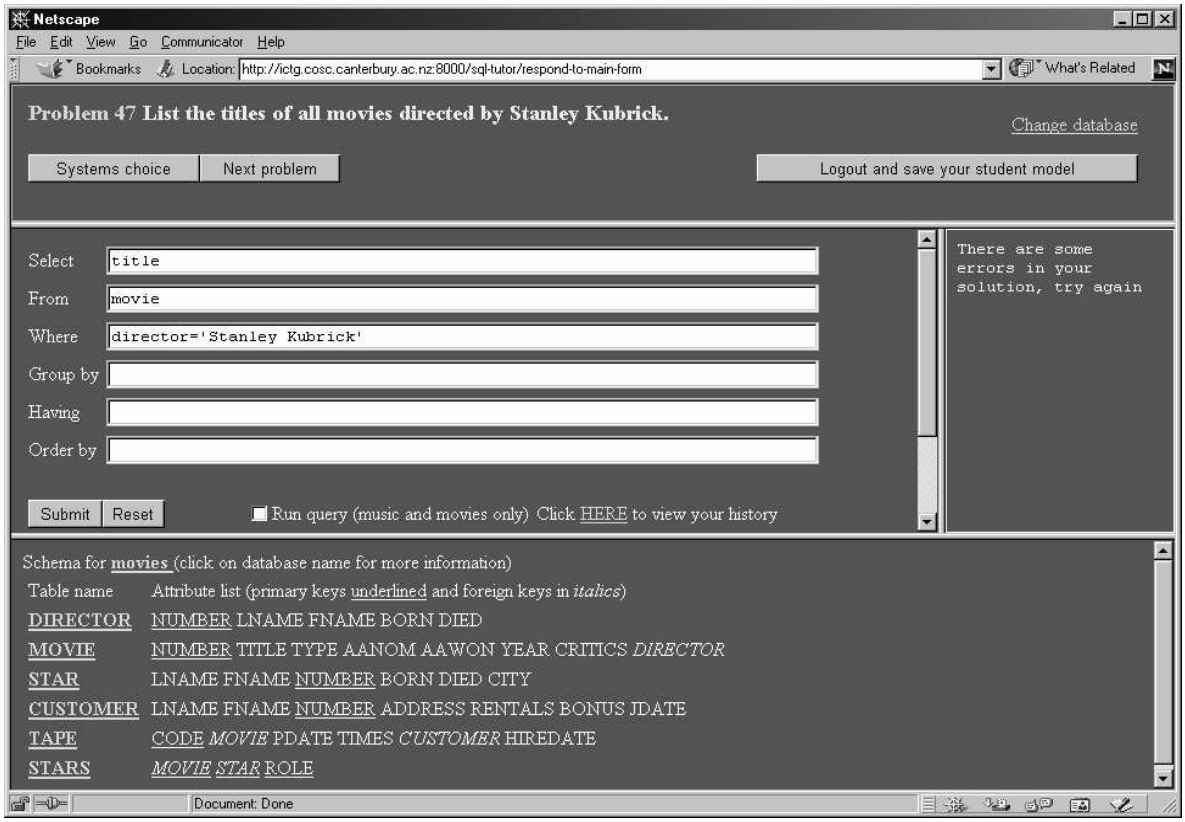

Figure 3. SQL-Tutor interface (web-enabled version)

SQL-Tutor contains 509 such constraints.

SQL-Tutor teaches SQL by presenting the student with English descriptions of queries, for which they must write an SQL SELECT statement. The answer section of the interface is structured into fields for the six clauses of a SELECT statement: SELECT, FROM, WHERE, GROUP-BY, ORDER-BY and HAVING. The student types their answer directly into these fields. At any time, they may receive feedback on their answer by submitting it to the system. At this stage the answer is evaluated by the constraint evaluator, and returns feedback regarding the state of their solution.

There are six levels of feedback: “Feedback”, “Error log”, “Hint”, “Partial solution”, “All errors”, and “Complete solution”. “Feedback” simply informs them that they are right or wrong. “Error Log” indicates which of the six clauses the first error encountered is in. “Hint” presents the feedback message for the first violated constraint. “Partial solution” displays the ideal solution for the clause to which the first violated constraint relates. “All errors” lists the feedback messages for all violated constraints. Finally, “Complete solution” simply displays the ideal solution in its entirety. The feedback level is automatically set by the system, and increments from “Feedback” to “Error log” to “Hint” automatically if the student continues to submit an incorrect answer. However, the student may override this behaviour by manually selecting the feedback type they require.

SQL-Tutor also attempts to ease cognitive load by providing *scaffolding.* The bottom section of the screen details the structure of the database the student is currently working on, so that they do not need to remember the details of the database tables, nor interrupt their work seeking help. This information may be drilled into if necessary for further detail.

The constraint set for SQL-Tutor (like all CBM tutors) is a flat set of constraints. Approximately half are semantic, and the other half syntactic. The constraints are unevenly distributed among the six SQL clauses: 12% are for the SELECT clause, 9% FROM, 34% WHERE, 32% HAVING, 6% GROUP BY and 7% ORDER BY. In particular, two thirds of the constraints are related to restrictions upon the data extracted from the database (i.e. the GROUP BY and HAVING clauses). The domain can also be split into basic queries (requiring a SELECT, FROM and WHERE clause)

and more advanced one, which also require the GROUP BY, HAVING, and ORDER BY clauses. The basic queries account for 55% of the constraint set, although note that the HAVING constraints are mostly identical to those for WHERE. Another way of splitting the constraints into basic and difficult is to consider those constraints concerning nested queries as advanced (approximately 10%).

In SQL-Tutor a single pedagogical state may be represented by more than one constraint. For example, there are 14 constraints with the feedback message:

```
"Check the integer constant you used with the aggregate function in
HAVING"
```

Of the 509 constraints, there are 347 distinct feedback messages. Also, there are cases where the same constraint is repeated because it is relevant to all clauses. Sometimes this is pedagogically significant, but in other cases it is merely a consequence of the way the constraints are encoded on a clause-by-clause basis. For example, the following constraint is present for all six clauses:

```
"You have ended the <clause-name> clause with a comma - that is not
allowed."
```

If feedback messages that are identical except for the clause name are considered equivalent, the number of pedagogically significant states further drops to 202. A further consideration is that up to a large number of the constraints deal explicitly with constructs where attribute names have been "qualified" by adding the table name, e.g. "MOVIE.DIRECTOR". This is really only a single concept: either the student knows how (and when) to use them or she does not. Also, many constraints test for the *absence* of a particular construct, for example:

```
'(p 95
 "Scalar functions (numeric, date or string ones) cannot be used in
the FROM clause - they may only appear in WHERE, HAVING and SELECT."

 t

 (null (intersection '("ABS" "DATE") (from-clause ss) :test 'equalp))

 "FROM")
```

The data from the study described in section 6.2 suggests that a student will cover around 25% of the constraint set after an extensive session with SQL-Tutor, and that this is sufficient to display proficiency in SQL.

CBM does not require any explicit structure to the model, however there may benefits to applying one: the model may be easier to maintain, and the model might form the basis of a curriculum. Adding structure might also allow the model to be opened to the teacher and student, by providing a high-level view of the student's performance. It also aids the selection of a target constraint for selecting the next problem. The difficulty is in deciding what structure to use, since this may differ according to how it will be used. Further, teachers may disagree on the structure, and students may learn the domain in different ways, necessitating individual domain structures. This is discussed in section 6.2.

SQL is an example of a declarative domain: the student's task is to transform a natural language description of a query into the SQL representation. The order in which they do this is not important. SQL is a relatively small language, because it is very compact: unlike more general programming languages such as Java, a single construct, such as a join in the FROM clause, has high semantic meaning, in that it implies considerable activity which is hidden from the writer (lock the table, open an input stream, retrieve the first record…). In spite of its syntactic simplicity, students find SQL very difficult to learn. In particular, they struggle to understand when to apply a particular construct, such as GROUP BY or nested queries. The major tasks of the tutor are therefore twofold:

1. To provide a rich set of problems, requiring many different constructs, that the student may learn when to apply them, and;
2. To provide drill in building those constructs.

SQL therefore seems well suited to CBM: given sufficient practise, the student will internalise *when* to apply each construct, and *how* to build it.

SQL-Tutor has a fairly large constraint set because of the amount of redundancy in the SQL language: there are often different ways to solve the same problem, and the details can vary greatly (e.g. qualification of attribute names, aliasing of table and/or attribute names). This gives SQL a quite high branching factor. However, more general programming tasks such as Java would probably be worse in this respect,

leading to very large constraint sets. An alternative might be to limit what the student can do. The latter is a common strategy in the domain of programming languages (Deek and McHugh 1998).

The response from students has been very positive, and statistical analysis of their performance indicates a significant improvement after as little as two hours of exposure to the tutor (Mitrovic and Ohlsson 1999). SQL-Tutor is now used regularly as part of a second year course on databases, and is popular with students. It is also the test bed for further research into ITS and CBM, including animated pedagogical agents (Suraweera and Mitrovic 2000), Bayesian student modelling (Mayo and Mitrovic 2000; Mayo and Mitrovic 2001), evaluating feedback effectiveness (Mitrovic and Martin 2000; Mitrovic, Martin and Mayo 2002), and the research described in this thesis (Martin 1999; Martin 2000; Martin and Mitrovic 2000a; Martin and Mitrovic 2000b; Martin and Mitrovic 2001a; Martin and Mitrovic 2001b; Martin and Mitrovic 2002a; Martin and Mitrovic 2002b).

3 Addressing limitations of CBM

Tutors built using CBM have been shown to be effective teaching environments. SQL-Tutor has been successful in the classroom, and is well liked by the students who use it. Further, the Web version is used by a large number of people world-wide: over 400 people have tried it, and when the Web version was last taken "off-air" for evaluation testing this invoked scores of emails from users wanting to know why it was no longer available. KERMIT elicited favourable qualitative feedback from students at Canterbury University. CAPIT, an ITS for teaching punctuation and capitalisation to elementary school children, has also been well received (Mayo and Mitrovic 2001).

In spite of these successes, there are still many improvements that could be made. We are interested in two main themes: increasing the usefulness of the knowledge base, and making constraint-based tutors easier to build. From these broad categories, we chose three specific goals to direct our enhancements to CBM: improving the quality of feedback, facilitating the generation and selection of new problems, and simplifying the creation of the knowledge base. Each of these is now introduced.

3.1 Feedback can be misleading

Feedback in SQL-Tutor is applied directly to each constraint in the domain model: when a constraint is violated, it produces a message that describes the underlying domain principle that has been failed. The student may additionally be shown all or part of a correct solution. SQL-Tutor selects problems from an authored set of examples. Each problem consists of the problem text, and an "ideal" solution to the problem. In SQL there is usually more than one correct query for any problem, so the ideal solution represents just one of a (possibly large) set of correct solutions. Because

the domain model is state-based, it is able to cope with differences between the student and ideal solutions by modelling the various different ways that a state (e.g. all tables used) might be represented.

However, problems arise when the ideal student solution is presented to the student as feedback. Through a series of in-class evaluations of SQL-Tutor, we have measured the apparent speed of learning while pupils interact with the system (Mitrovic and Martin 2000). Analysis of the data obtained indicates that differences in the feedback given have a significant effect on the speed of learning. For example, presenting either the constraint feedback or part of the ideal solution increases learning speed, while presenting the entire solution is detrimental. Further, one of the most successful modes of feedback is “partial solution”, where the pupil is presented with the fragment from the ideal solution for one of the SQL clauses in which they have made mistakes. The drawback with this approach is that the fragment may sometimes be correct within the context of the ideal solution, but *incorrect* within the context of the student solution. Consider the following example:

Problem:

List the titles of all movies directed by Stanley Kubrick.

Ideal Solution:

```
SELECT   title
FROM     movie
WHERE    director=(select number from director
           where fname='Stanley' and lname='Kubrick')
```

Student Solution:

```
SELECT   title
FROM     movie join director on number = director
WHERE    fname='Stanley' and lname='Kubrick'
```

Submitting this attempt (correctly) yields the following result:

"You have used an attribute in the FROM clause which appears in several tables you specified in the FROM clause. Qualify the attribute name with the table name to eliminate ambiguities."

However, when prompted for a partial solution for the FROM clause, the system returns “FROM movie”, which is correct for the ideal solution but *not* correct within the context of the student solution.

A CBM knowledge base uses the constraints to diagnose the student’s answers and build a student model, the latter being described by Ohlsson only in passing. In contrast, Cognitive tutors are able to indicate to the student what they should be doing *next*, by modelling the problem-solving steps. While a CBM-based system can comment on the incorrect solution (even if empty), it cannot offer specific instructions on how to fix errors because the model intentionally contains no information whatsoever about the problem-solving *procedure*. Neither is CBM able to solve the problem in order to indicate a possible solution, because the knowledge elements (constraints) are discrete and so lack information indicating the sequence in which they should be considered. Finally, the only requirement of each constraint is that they are able to test whether or not the student solution is in the appropriate relevance and satisfaction states, but they do not need to be able to indicate *why* the answer is in this state or *how* to get it into this (satisfaction) state if it is not there already.

In the three CBM implementations described earlier (SQL-Tutor, CAPIT, and KERMIT), this limitation is real. In SQL-Tutor, constraints are encoded in LISP. In each constraint the relevance and satisfaction conditions are each a standard LISP condition consisting of function calls, combined using the standard LISP logical connectives AND, OR and NOT. Functions may be internal to LISP (e.g. CONS, FIRST) or domain-specific (e.g. `attribute-of(table, attribute)`, which tests whether a particular term *attribute* is belongs to the database entity *table*). A domain-independent pattern-matching routine is also included. Similarly, KERMIT contains a domain-specific functional language for representing constraints. CAPIT (Mayo and Mitrovic 2001) uses an extension of regular expressions to represent constraints. It is possible that this could be used to correct errors, although no attempt is made to do so. However, CAPIT is unusual in that the domain is deterministic: there is only one way to correctly capitalise and punctuate a sentence in New Zealand English. Further, the domain is simple: there are only 25 constraints. It is doubtful that the language used in CAPIT could be extended to more complex domains because it is too limited.

The consequence of this is that none of these systems can tell the user what their answer *should have been* because when they discover a constraint violation, they cannot determine what to do about it. In SQL-Tutor the output of the constraint evaluation process is the list of satisfied and violated constraints, and the variable bindings for all instances where each constraint was found to be relevant. This is of little use however, since the system is unable to use them to run the constraint evaluation *in reverse* to arrive at a correct version (with respect to this constraint) of the student solution. KERMIT also uses a domain-specific representation of constraints, which include specialised sub functions plus ad hoc algorithms to evaluate solutions against the constraints.

KERMIT, CAPIT and SQL-Tutor therefore not only lack an algorithm for problem solving, for two of them it would not be possible directly from the constraint set because the constraints do not provide the information necessary to overcome a violation. To be able to correct errors, it would be necessary for these systems to include specific "repair" functions for each constraint, which would (at least) double the constraint authoring effort. However, since each constraint maps a relevance state to a satisfaction state, it in some way describes how the student solution, if in the relevance state, should be further constrained in order to satisfy the underlying declarative rule. It therefore seems that the constraint *does* encapsulate how to produce a valid part-answer from a certain range of inputs (i.e. those relevant to this constraint), but this information is hidden in the functions that carry out the testing. Therefore, what is needed is a constraint representation that makes all of this information transparent, and hence able to be used by an algorithm to generate a part-solution that satisfies the constraint. We have developed such a representation, which is described in Chapter 4.

Once the constraints are in a suitable representation, an algorithm is needed that reverses the logic of the constraint evaluator, i.e. given a satisfaction condition it produces a part-solution that satisfies it with respect to the ideal solution and the student's attempt. We have developed such an algorithm, which is described in Chapter 5.

3.2 Limited problem set

Another feature the three ICTG systems share is that they contain enumerated problem sets that have been hand-authored by their creators. SQL-Tutor currently contains 82 such problems, KERMIT has six, and CAPIT has 45. Each system chooses problems based on the student model: the constraints which are relevant to each question are compared to the student model, to see how they match the student's performance on those constraints. The problem that best matches the currently targeted constraints wins. Different methods are used to determine the target constraint(s). SQL-Tutor uses the constraint that has been violated most often. This may be selected from the entire constraint set, or from a subset from a specific part of the curriculum, such as "sorting". In deciding which constraint has been violated most frequently, either all or a recent subset of the student's behaviour may be considered. Candidate problems are those for which this constraint is relevant, with the problem of the most suitable difficulty being selected. A similar system is used in KERMIT. CAPIT, in contrast, uses a Bayesian normative system to decide which constraint should be targeted and to determine which problem is most suitable.

The problem with all of these systems is that the size of the problem set is very limited, and so although the problem selection criteria may appear sensible, in practice they may of limited use because there are so few problems to select from. For example, in SQL-Tutor each constraint is, on average, relevant to three problems in the set, with only 20% of the constraints being relevant to any problem at all. It is therefore highly likely that if a student is having particular trouble with an individual constraint, they will quickly exhaust all relevant problems. Further, since not all constraints have relevant problems, there are many concepts in our domain knowledge base that we are simply unable to test. We also need to be able to support users of varying (and changing) abilities, so we need to be able to test each constraint using problems that have varying difficulties, which further increases the number of problems required. Finally, in determining the difficulty of a problem with respect to the current student *at this moment in time*, we should ideally consider the student's knowledge of each constraint relevant to each problem. This means the real range of difficulties a problem set represents *changes with time*, so we cannot guarantee at any

time that we have a problem of a given difficulty with respect to the student. For example, if a student does very well and rapidly learns a large proportion of the constraints, the difficulty of the problems with respect to that student may quickly drop too low to contain any problems that are sufficiently challenging. Conversely, if they keep violating constraints, the simplest problems may soon appear beyond their ability. This increases the range of difficulties we need to represent, and so further increases the number of problems required.

None of these issues are caused by CBM per se: to reduce them we simply need to write more problems. However, writing problems is hard. It took many days to write the 82 problems for SQL-Tutor. Moreover, it is difficult to write a problem set that covers all constraints: we can either write problems independently of the constraints and regularly test them against the constraint set to see what the current coverage is, or we can manually investigate each constraint, and try to write several problems that cover that constraint over a range of difficulties. Recalling that SQL-Tutor contains over 500 constraints, this will be a large undertaking.

In the previous section we argued that it should be possible to use the information in the constraints to correct a student answer with respect to an individual constraint. A special case is when the student answer is blank. In this case, we are solving a problem from scratch. Recall that in all three ICTG systems, an ideal solution is used when diagnosing the student answer: both the student answer and the ideal solution are used as input to the constraint evaluator. However, for around half of the constraints, only the syntax of the student answer is tested, and so the ideal solution is not required. Since constraints are modular, it follows that these can act independently of the semantic constraints to determine whether the current solution is valid SQL, regardless of the ideal solution. If, as suggested in the previous section, the constraints can map an incorrect solution to a correct one, the syntactic constraints should be able to map an invalid SQL statement to a valid one. Given a suitable starting point, it should be possible to automatically generate an arbitrary SQL statement. In Chapter 6 we show how we can use an individual constraint as the starting point and automatically “grow” a valid SQL statement for which this constraint is relevant, and we describe an algorithm we have developed for doing this. This new SQL statement forms the ideal solution of a new problem for testing this constraint. We can then

apply this algorithm to the entire constraint set, to generate problems to cover the domain. Thus while CBM does not cause the limitations discussed in this section, it can be used to overcome them.

3.3 Building an ITS is hard

CBM knowledge bases are easier to build for some domains than those adopting model tracing. For example, the knowledge base in KERMIT contains just 92 constraints. Recall from Section 2.4.3 that an equivalent representation for a Cognitive tutor required a total of 55 non-trivial knowledge elements to represent just 23 constraints and two problem statements (with 11 tags). While direct comparisons can be misleading, this nevertheless seems a significant difference. Further, in Cognitive tutors each knowledge element has an effect on a potentially large region of the model. In comparison, modifying a constraint has no effect at all on the rest of the constraint set.

In Cognitive tutors the model has a high requirement of fidelity. An incomplete model will prevent the user completing the task. Thus, having decided what problems you want the student to solve, a complete model must then be built for the procedures involved. In contrast, CBM does not have this necessity. An incomplete constraint-based model will fail to catch some student errors but it will not prevent the user finishing the problem. A missing constraint is therefore not a catastrophic situation, unlike a missing knowledge element (or procedural rule) in Cognitive tutors. This enables CBMs to be built incrementally, adding new constraints whenever incorrect student answers are “let through” by the system.

In spite of these advantages CBMs are still hard to build. It is unrealistic to expect someone other than an ITS engineer to build one from scratch since they require a good knowledge of how the system will use the constraints, as well as of the domain itself and the representation used to build the knowledge elements. Since, in Section 3.1, we identified a need for a new constraint representation, we also took this opportunity to develop a *simple* constraint representation, together with a set of rules on how constraints should be written, so that someone other than an ITS engineer (e.g. a domain expert) might be able to produce the knowledge base for a given

domain. This is discussed in Section 7.3. A consequence of the new representation and its requirements to facilitate problem solving is that all domain-specific information must be explicitly encoded in the constraints. This has the advantage that the domain-specific information is cleanly separated from the constraint evaluator. We can capitalise on this feature to produce an authoring tool for constraint-based ITS. We have developed such a system, Web-Enabled Tutor Authoring System (WETAS), which is described in Chapter 7.

3.4 Summary

CBM is a relatively new approach to domain and student modelling that simplifies the construction of domain models. In doing so however, it loses one of the major advantages of other methods such as model tracing: it is unable to indicate what the student *should have done*. As a result, it provides a poorer level of feedback. We have sought to overcome this limitation. We have also endeavoured to address the continuing problem of how to make ITS easier to construct. In the next four chapters, we describe these enhancements to CBM.

4 Constraint representation

To be able to reason about the nature of constraint violations, either every operation performed during constraint evaluation must have a corresponding counter-operation that corrects the fault, or all of the information used to test the constraint must be available to the problem-solving algorithm. The latter has the benefit of (theoretically) not requiring any more effort on the part of the author, whereas the former would roughly double the knowledge engineering effort. For this reason, we introduce a new representation of constraints is more transparent, and is *reversible*. Reversible means that a constraint defined in this language can not only be used to determine whether or not the solution satisfies it, it can also be used to *enumerate correct solutions* with respect to this constraint.

The constraint representation in SQL-Tutor already uses pattern matching via the domain-independent MATCH function. However, it also uses many domain-specific functions to decide valid values of terms, test for compatibility of two or more values (e.g. an attribute and the table that the attribute appears to belong to), and to post-parse terms such as qualified names. These functions hide the logic that determines whether or not a solution is correct. For example, the function `valid-table(t1)` determines whether or not `t1` is a valid table name, but is unable to tell us what `t1` *should be* if this test fails. It is these functions that must be removed from the representation. We investigate whether this can be overcome, and propose the following hypothesis:

> **Hypothesis 1**: It is possible to build a constraint-based domain model that contains sufficient information to solve problems and correct student solutions, by adopting a constraint representation that makes all of the logic in each constraint transparent to the system.

4.1 Constraint representation

In the new representation, constraints are encoded purely as pattern matches. Each pattern may be compared either against the ideal or student solutions (via the MATCH function) or against a variable (via the TEST and TEST_SYMBOL functions) whose value has been determined in a prior match. An example of a constraint in SQL-Tutor using this representation is:

```
(34
"If there is an ANY or ALL predicate in the WHERE clause, then the
attribute in question must be of the same type as the only
expression of the SELECT clause of the subquery."

; relevance condition
(match SS WHERE (?* ?a1
  ("<" ">" "=" "!=" "<>" "<=" ">=")
  ("ANY" "ALL") "(" "SELECT" ?a2 "FROM" ?* ")" ?*))

; satisfaction condition
(and (test SS (^type (?a1 ?type))
     (test SS (^type (?a2 ?type))))

"WHERE"
)
```

This constraint tests that if an attribute is compared to the result of a nested SELECT, the attribute being compared and that which the SELECT returns have the same type (`^type` is an example of a macro, which are described in section 4.1.4). The new representation consists of logical connectives (AND, OR and NOT) and three functions: MATCH, TEST, and TEST_SYMBOL. These are now described.

4.1.1 MATCH

This function is used to match an arbitrary number of terms to a clause in the student or ideal solutions. The syntax is:

```
(MATCH <solution name> <clause name> (pattern list))
```

where `<solution name>` is either `SS` (student solution) or `IS` (ideal solution) and `<clause name>` is the name of the SQL clause to which the pattern applies. However, the notion of clauses is not domain-dependent; it simply allows the solution

to be broken into subsets of the whole solution. The `(pattern list)` is a set of terms that match to individual elements in the solution being tested. The following constructs are supported:

- `?*` – **wildcard**: matches zero or more terms that we are not interested in. For example, `(MATCH SS WHERE (?* ?a ?*))` matches to any term in the WHERE clause of the student solution, because the two wildcards can map to any number of terms before and after `?a`, so all possible bindings of this match gives `?a` bound to each of the terms in the input;

- `?*var` – **named wildcard**: a wildcard that appears more than once, hence is assigned a variable name to ensure consistency. For example:

```
(AND (MATCH SS SELECT (?*w1 "AS" ?*w2)          (1)
     (MATCH IS SELECT (?*W1 ?N ?*))             (2)
```

  First, (1) tests that the SELECT clause in the student solution contains the term `"AS"`. Then, `?*W1` in (2) tests that the ideal solution also contains all the terms that preceded the `"AS"`, and then maps the variable `?N` to whatever comes next. The unnamed wildcard at the end of the second MATCH discards whatever comes after `?N`;

- `?var` – **variable**: matches a single term. For example,

```
(MATCH IS SELECT (?what))
```

  matches `?what` to one and only one item in the SELECT clause of the ideal solution;

- `"str"` – **literal string**: matches a single term to a literal value. For example, in

```
(MATCH SS WHERE (?* ">=" ?*))
```

  one of the terms in the WHERE clause of the ideal solution must match exactly to `">="`;

- `(lit1 lit2 lit3...)` – **literal list:** list of possible allowed values for a single term. For example:

```
(MATCH SS WHERE (?* ?n1 (">=" "<=") ?n2 ?*))
```

assigns the variable `?n1` to any term preceding either a `">="` or a `"<="`, and `?n2` to the term following it. Note that because `?n1` and `?n2` are not wildcards, they must map to a single term each, hence if the `">="` or `"<="` is either at the start or the end of the clause this match will fail, because one (or both) of `?n1` and `?n2` will fail to match.

Variables and literals (or lists of literals) may be combined to give a variable whose allowed value is restricted. For example,

```
(MATCH IS ORDER_BY (?* (("ANY" "ALL") ?what) ?*))
```

means that the term that the variable `?what` matches to must have a value of `"ANY"` or `"ALL"`. There is no limit to the number of terms that may appear in a literal list, or in a MATCH in general.

4.1.2 TEST

Having performed a MATCH to determine the existence of some sequence of terms, we often wish to further test the value of one or more variables that were bound. This is carried out using the TEST function, which is a special form of MATCH that accepts a single pattern term and one or more variables. For example (the following constraint is simplified):

```
(2726
"Check you have used the correct logical connective in WHERE to
represent a range of numbers."

(and
    (match SS WHERE (?* ?n1 ?op1 ?what1
                    (("and" "or") ?lc)
                    ?n1 ?op2 ?what2) ?*)                (1)
    (match IS WHERE (?* ?n1 "between" ?*))              (2)
)

(test SS ("and" ?lc))                                   (3)

"WHERE"
)
```

This constraint first tests for an attribute (`?n1`) in the WHERE condition of the student solution that is being compared to two different values (`?what1` and `?what2`) in (1). Then, (2) looks for the same attribute being used in a BETWEEN construct in the ideal solution. If this is the case, the two tests in the student solution must be ANDed together. The TEST function call in (3) checks that this is the case, by ensuring that the logical connective (represented by the variable `?lc`) equates to "and". The syntax of the TEST function is:

```
(TEST <solution name> (test-term))
```

where `<solution name>` is again IS or SS, and `(test-term)` is a single value test, such as a test against a literal or list of literals. In the previous example, a single value test is made for the value `"and"`. In effect, TEST performs the same function as MATCH, but where the pattern contains just a single match term, on a list that contains just the value of the variable in question, in this case `?lc`.

4.1.3 TEST_SYMBOL

We often need also to be able to test characters within the value of a term. For example, a valid SQL string is defined as a single quote followed by any characters, and closed with another single quote. To test this we add the function TEST-SYMBOL, which acts exactly like the MATCH function, except it accepts a variable name instead of a clause name, and further parses the value of the variable binding into individual characters, before applying the match pattern. For example, to test for a valid SQL string in the variable `?str`:

```
(TEST_SYMBOL SS ?str ("'" ?* "'"))
```

This test would succeed for values of `?str` such as `"'Kubrick'"` for example, but fail for `"'Smith"` because of the missing closing quote. The general syntax is:

```
(TEST_SYMBOL <solution> <var> (pattern))
```

Note that in both TEST and TEST_SYMBOL, the solution name is passed as a parameter even though it doesn't appear to be necessary, since these tests are on already bound variables, not an input string. However, this is required because the test

may be a *macro*, which may perform further pattern matches on the input, so it needs to know which solution to match. Macros are now described.

4.1.4 Removing domain-specific functions: macros

At the start of this chapter we stated that SQL-Tutor uses domain-specific functions to extract features of the solutions and to make special comparisons between them. In the new representation this is forbidden, because it hides the logic of the test. In SQL-Tutor almost all domain-specific functions test for a valid value or pair of values. For example, in:

```
(valid-table (find-schema (current-database *student*)) '?t1)
```

`Valid-table` tests that `?t1` is a valid table name in the student's current database. Similarly:

```
(attribute-of (find-table '?t1 (current-database *student*)) '?a1))
```

tests that `?a1` is a valid attribute in the table `?t1`. Routines such as `find-table` and `current-database` (a Common Lisp Object System selector method) are simply data accessors. In both `valid-table` and `attribute-of`, the function might alternatively be represented as a membership test on an enumerated list: for `valid-table` the list will contain the set of table names for a given database, while for `attribute-of` each member of the list will be a tuple of type `(<attribute> <table>)`. Since our language already supports testing against lists of literals, these can be encoded using the pattern matching language, i.e.

```
(TEST SS (("MOVIE" "DIRECTOR"...) ?t1))
```

which tests that `?t1` is a valid table, and

```
(TEST SS (("TITLE" "MOVIE") ("YEAR" "MOVIE") ("LNAME"
"DIRECTOR")...)
          (?a1 ?t1))
```

which tests that `?a1` and `?t1` form a valid attribute/table combination, i.e. that `?a1` is an attribute of table `?t1`.

Many of the other domain-specific functions are either accessor functions or perform pattern matching. The former can be eliminated by making the required

values available to the pattern-matching algorithm (for example, as standardised global variables), while the latter can all be achieved using the pattern matching language itself. The only function from SQL-Tutor we were unable to represent elegantly was “length”, which tests how many of a particular item exist. In practice, we could represent this function in our language by some other means in all but one case of its use. In the single exception, which tests for the number of attributes in the SELECT clause, we did not consider this constraint pedagogically necessary, although we could have easily coded it via a macro that enumerates all possibilities. However, it remains an open question whether other functions (such as a more general “length” function) would be necessary in other domains.

We have now replaced function calls with pattern matching, however it would be cumbersome to have to enumerate all attributes of all tables every time we wish to perform such a test. To overcome this we use macros to represent partial pattern matches that are used often. For example, the macro for `^attribute-of` used previously is:

```
(^attribute-of (??a ??t)
    (TEST SS ((("TITLE" "MOVIE") ("LNAME" "DIRECTOR")...)
              (?a1 ?t1)
             )
    )
)
```

The syntax of a macro definition is:

```
(<MACRO NAME> (<parameters>) <body>)
```

The name must always begin with a “^” so that macros can be easily identified by the constraint compiler. Similarly, the parameter names are preceded by “??” so that they can be distinguished from local variables in the macro body. The body may be any valid condition including logical connectives, MATCH functions and other macro calls. Consider the following example from SQL:

```
(^attribute-alias (??name ??attr ??table)
  (and
    (test ?? (^name ??name))
    (or-p
      (test ?? (^attr-name (??name ??attr ??table)))              (1)
      (match ?? SELECT
           (?* (^attr-name (?_a1 ??attr ??table)) "AS" ??name))   (2)
```

```
    )
   )
)
```

This macro accepts an attribute name as input and returns the physical attribute and table names. In SQL attributes can be aliassed, i.e. they can be assigned another name. For example:

```
SELECT movie.number AS num
FROM movie
ORDER-BY num
```

In this example, "num" is defined as an alias for `movie.number` in the SELECT clause, and is used again in ORDER-BY. To test that num in ORDER-BY is a valid attribute, we need to know what it maps to, which is achieved by the `^attribute-alias` macro defined previously. If `??name` fails the test in (1), i.e. it is not a valid attribute name, (2) tries to match it to an alias definition in the SELECT clause. Hence, the macro needs to know which solution it is testing. The constraint that tests for a valid attribute in ORDER-BY is therefore:

```
(149
"You have used some names in the ORDER BY clause that are not from
this database."

(match SS ORDER_BY (?* (^name ?n) ?*))

(test SS (^attribute-alias (?n ?a ?t)))

"ORDER BY")
```

When the constraint set is loaded, the macro names are expanded into their corresponding pattern matches. The parameter names in the macro definition are substituted for those passed in, and the "??" solution name placeholders are replaced with the solution name from the caller. Hence, all routines that can call a macro (i.e., MATCH, TEST and TEST_SYMBOL) must specify a solution name. Note that macros may also be embedded in pattern matches, and that the macro being called may have more than one parameter. For example:

```
(match SS SELECT (?* (^attr-name (?n ?a ?t)) ?*))
```

In this case, the *first* parameter to `^attr-name` (`?n`) is matched to a term in the input string, with `?a` and `?t` being either tested or instantiated by the macro, depending on whether or not they are already bound.

When the constraints are compiled all macro calls are recursively removed such that the resulting code contains purely pattern matches with no sub functions, and hence all tests are fully enumerated. It is this property that facilitates generating SQL from the constraint set.

4.1.5 Limitations of the representation

The constraint language is limited by the need to be able to generate solutions that satisfy the constraint set. This means that it must be possible to enumerate the set of constructs that satisfies each constraint. This gives rise to two limitations: the inability to call external functions, and a lack of recursion. Each is now described.

Inability to call external functions

It is not permissible for constraints to call external functions, such as arithmetic operations, since these could not be relied upon to return the set of all possible answers in a given situation. For example, the built-in "+" function could not return all possible values of X and Y in the question "X + Y = 5?". This is a common problem with languages that perform unification: for example, PROLOG is unable to answer such questions. The workaround is to write these functions in the pattern matching language, usually by enumerating all cases in which we are interested. In SQL-Tutor, for example, the operator INCR (where INCR(X,Y) means Y = X + 1) was encoded by enumerating all values of X and Y that arose in the problem set.

Lack of recursion

A fundamental limitation of the new representation is that it cannot represent recursive definitions: although the macros add structure to the constraints in raw form, the expanded representation of any test is simply the logical connection of linear pattern matches upon a set of strings. This means that functions such as "greater_than" and "less_than", for example, would need to be enumerated, rather than writing a recursive definition. In the SQL domain, it was necessary to test

whether a number was one larger than another, and this too had to be enumerated. Perhaps more seriously, the domain itself may allow recursive constructs, such as nested loops. In SQL, queries may be nested in the SELECT clause to an arbitrary depth. It is not possible in the new constraint language to represent such constructs, and so it is not possible to test their correctness. However, it is still possible to represent a suitable subset. In SQL-Tutor, we set a limit of three levels of nesting, since we reasoned that it is unlikely a query would require more than this, so we would never set a problem that exceeded this limit. We then wrote constraints that explicitly tested for up to three levels, i.e. we enumerated all the possibilities from zero to three levels of nesting. However, there may be other domains for which recursion is more fundamental, and hence this limitation may be more difficult to overcome.

4.2 The constraint evaluator

The constraint evaluator performs three functions: test the student solution against the constraint set, extract relevant fragments of the solution, and collate the set of corrections that need to be performed (if any). These latter functions are required for problem solving (Chapter 5).

Constraints are evaluated one at a time. On completion, the solution is either correct with respect to the constraint set (i.e. it does not violate any constraints) or it has violated one or more constraints and may be passed on for correction. For each constraint the relevance condition is first checked. If it fails, the constraint is not relevant and no further action is taken. Otherwise, the satisfaction condition is checked. If this fails, the list of required corrections is passed on. If the constraint succeeded, binding and fragment information is recorded.

All of the individual statements in the relevance condition are evaluated using the pattern matcher until a failure occurs, which signifies that we are no longer interested in this constraint. However, when the satisfaction condition is evaluated, we test *all* bindings that resulted from the relevance condition even if a failure has been encountered, so that we have failure information about all of the failed bindings, not just the first. The overall constraint evaluation algorithm is:

```
Test-solution:
For each constraint in the set
   if the relevance condition evaluates
      For each set of valid bindings
         if the satisfaction condition evaluates
            Add the (modified) binding set to the set of
              fragment information for this constraint
         else
            Add the correction information for this binding set
              to the list of corrections needed for this
              constraint
      Add the binding and correction information for this
       constraint to the set for this problem.
```

Constraints are tested directly by evaluating the LISP fragments that they consist of. There are only six functions that may be called: MATCH, TEST, TEST_SYMBOL, AND, OR-P, and NOT-P. “AND” is the built-in LISP function, while OR-P and NOT-P are modified versions of the built-in functions OR and NOT, which are required to maintain binding and correction information consistency during failures. The logic pattern-matcher is therefore contained wholly within these functions, with MATCH and TEST being the most important. Further, all binding and correction information is collected by these functions. Each function is now described.

The MATCH and TEST Functions

The MATCH and TEST functions are essentially wrappers around the same algorithm. In the case of MATCH, the input is a clause from the student solution, which is tested against a list of one or more pattern terms. TEST accepts a single pattern term and has no other input: all variables participating in the pattern are assumed to be either instantiated already, or they will be instantiated as a side effect of the test.

When a MATCH is performed, some of the terms in the match pattern may have already been bound by previous matches. Because matching can include wildcarding, some terms may have already been bound *in multiple ways*. Therefore, an underlying function, MATCH-BINDINGS, takes the pattern list and tries to match it to the student solution for each valid set of bindings so far. On commencement of evaluating a given constraint, the binding set contains just a single set with a default root binding. Each time a variable is encountered, the binding set is updated with the current binding set being duplicated for each possible binding value for the variable.

Each of these new sets is then recursively tested against the rest of the pattern to try to complete the match. At any point one or more of these binding sets may be further split because another variable in the same pattern is encountered that can be satisfied in multiple ways. Further, during problem solving (see Chapter 5), each clause of the interim solution may contain more than one fragment, so MATCH-BINDINGS tests each fragment individually. The final binding list returned is a list of the sets of binding values for which the pattern successfully matches.

The MATCH function further modifies the binding set returned from MATCH-BINDINGS. Each time a match is successful, it implies that the associated piece of SQL is of interest to us for problem solving. Therefore, the pattern itself is recorded in each valid binding set. The problem solver can then instantiate the fragment with the bindings from the set to reconstruct the original fragment of the solution that is relevant to this constraint. Similarly, the TEST function records successful tests in the binding information. This is required because many tests result in new variables being instantiated, which are later involved in a MATCH or TEST. Without this information, the link between the variables would be lost, so corrections made to the latter variable because of a failure would not propagate back to the original MATCH. Consider the following constraint:

```
(21
"If a DATE type attribute is used in a condition, it must be
compared to an attribute of the same type."

(and   (match SS WHERE (?* (^attribute-p (?a ?att ?t)) "=" ?c ?*))
       (test SS (^type-p (?att "DATE")))
)

(and   (test SS (^attribute-p (?c ?att2 ?t2)))            (1)
       (test SS (^type-p (?att2 "DATE")))                 (2)
)

"WHERE")
```

When `?c` is tested in (1) to see if it is in fact an attribute, a side effect is that `?att2` and `?t` are instantiated to the physical attribute name and table name that `?c` represents. If (2) fails, `?att2`, which is an intermediate variable only, is corrected. Test (1) therefore needs to be recorded so that the problem-solver can deduce that `?c`

also needs to be corrected because `?att2` was derived from it. An example of a correction list entry with such a test is given in Section 5.7.1.

Logical Connectives

We are prevented from using the inbuilt OR and NOT functions by the fact that MATCH and TEST update the binding lists as they are encountered. Therefore, if a disjunct consists of a conjunction of MATCH and TEST calls, some of these may succeed before the disjunct as a whole finally fails, and so the binding list will contain fragment entries for part of the failed disjuncts as well as for successful ones. This is also true in the case of NOT: if the test being negated itself succeeds, it will update the fragment list, even though this means the negation has failed. The custom routines OR-P and NOT-P ensure that the binding set is consistent, by restoring it each time a disjunct or negation fails.

The pattern matcher

The MATCH, TEST and TEST_SYMBOL functions share a common pattern matcher, which tries to match a given pattern list to an input fragment one term at a time. The matcher works from left to right, maintaining the current binding set as it goes. The pattern terms may each be one of an unnamed wildcard, named wildcard, literal, or variable. Each of these is treated differently.

Literals (and lists of literals) are the simplest type of pattern term: the next term must match the literal exactly. Variables are more complex. If the variable hasn't been instantiated yet, it may take any value. If it has been instantiated, the next term must match the instantiated value. Further, the variable term may also contain a match (e.g. a list of allowed values) that must be met. The term is first compared with any match requirements, and then compared to the current binding set to ensure consistency is maintained.

The purpose of an unnamed wildcard is to "consume" zero or more terms until the rest of the pattern can succeed. When one is encountered a flag is set that indicates that if a subsequent term fails, the matcher may backtrack to this position, drop the current input term and try again. In contrast, the behaviour for a named wildcard depends on whether or not it is instantiated. If the wildcard variable is not currently instantiated, it behaves the same as an unnamed one except the binding list is updated

with an entry for the wildcard on successful completion of the pattern. However, if the wildcard has been already instantiated, it is treated the same as a literal: the next *n* terms must now be the same as the value of the wildcard, where *n* is the length of the original match to this wildcard.

Because a variable may be preceded by a wildcard, it can potentially take more than one value, which will cause the current binding set to be split into many. Further, there may be more variables further on in the pattern that allow this branching to happen again. Each variable value must therefore be resolved recursively for the rest of pattern: if the rest of the pattern succeeds, one or more binding sets are inserted into the binding list representing all the ways the pattern could resolve given the current value for this variable. Then, if the variable was preceded by a wildcard, the next potential value is obtained, and again the rest of the pattern is tested to see whether it returns any further valid bindings.

Finally, special consideration must be given to the situation where either the pattern list or the input runs out before its counterpart. The input is only permitted to run out before the pattern list if the remainder of the pattern list consists entirely of unnamed wildcards, un-instantiated wildcards, and wildcards instantiated to NULL. Conversely, the pattern is only permitted to run out before the input if the last term is a wildcard that can be resolved to the remainder of the list.

4.3 Summary

We have developed a new constraint representation where all testing functions are transparent and reversible, and implemented the associated constraint evaluator. We have reimplemented the domain model for SQL-Tutor using the new representation. We use this version of the domain model and constraint evaluator in the classroom evaluation in Chapter 6, which demonstrates that it works. Recall that hypothesis 1 required a transparent representation that facilitates problem solving. We have now defined the representation and shown it to be feasible for at least the SQL domain. In the next chapter we show that it is sufficient to solve problems in this domain.

5 Problem solving using constraints

The CBM approach obviates the need for a problem solver because the constraints are only interested in the solution state and can check this by testing the student solution directly against an ideal solution. Strategy variation is allowed for within the constraints by testing at the conceptual level (e.g. does this solution have all the necessary tables represented *somehow*, rather does this solution represent tables the *same way as the ideal solution*). Even procedural domains can be represented this way by capturing declarative knowledge that constrains sequences of events ("you must have started the engine before you release the clutch").

However, CBM does not *preclude* the use of a problem solver. In this chapter we explain why one is beneficial and describe the implementation of a problem-solving algorithm that uses just the existing constraints to arrive at correct solutions.

5.1 Motivation

We described in Chapter 3 how feedback could be misleading. In many domains including SQL, there is more than one way to solve a problem. There can therefore be valid differences between the student and ideal solutions. Often these will be minor, such as performing two unconstrained tasks in a different sequence (in the case of a procedural domain), or using a qualified name instead of an unqualified one in SQL. In other cases however, the entire problem solving *strategy* may differ. Recall the example from Chapter 3:

Problem:
List the titles of all movies directed by Stanley Kubrick.

Ideal Solution:

```
SELECT   title
FROM     movie
WHERE    director=(select number from director
           where fname='Stanley' and lname='Kubrick')
```

Student Solution:

```
SELECT   title
FROM     movie join director on number = director
WHERE    fname='Stanley' and lname='Kubrick'
```

The ideal solution uses a nested SELECT to obtain supplementary data from a second table, while the student uses a JOIN, which is a totally different strategy to solving the problem. This doesn't pose a problem until the student is presented with part or all of the answer as feedback. In the above case the full solution is of no use to the student unless they are prepared to abandon their attempt, in which case they do not get to complete the learning they are currently experiencing. However, to be shown a partial solution is worse: both the FROM and WHERE clauses of the ideal solution would be wrong in the context of the student's attempt. Since we believe that showing a partial solution is beneficial (Mitrovic and Martin 2000), we need to address this shortcoming.

In a model tracing ITS we might try to get from the erroneous solution back onto a correct solution path by either using a bug library to determine what is wrong, or retracing the sequence of steps back to where the student solution first deviated from a correct path. In the case of CBM, we do not have a bug library, nor do we have any method of getting back to a desirable solution. We therefore desire a problem solver that uses the constraints themselves to solve the problem. Moreover, since the student solution may be incorrect in any number of ways that we have never seen before, we would like this problem solver to be able to arrive at a correct solution given an arbitrary student solution. To be useful the resulting solution should be as close to the student's attempt as possible.

Determining how to track and understand students' (sometimes incorrect) problem-solving procedures remains an important problem in ITS research. This is particularly evident in the complex domain of programming. Various approaches have been tried, but (Deek and McHugh 1998) report that almost all of them constrict the student's freedom in some way. The main issue is determining the student's *intent*,

such that bugs can be understood and corrected in a logical way. Model-tracing tutors overcome this problem by forcing the student to stay very close to one or more "optimal" solution paths. Since building up these paths is difficult, often only one is provided. The LISP Tutor (Anderson, Farrell and Sauers 1984; Anderson and Reiser 1985) relies on a bug catalogue, which models divergence from the expert behaviour to keep the student within one step of the solution path so that the tutor always knows their intent. This, combined with the language-sensitive-editor style of the user interface, ensures that the system is always able to complete the solution by simply carrying out the rest of the model. The ACT Programming Tutor (Corbett and Anderson 1993) similarly models "ideal" solution paths. However, model tracing does not guarantee that student errors can always be corrected. Sometimes a student may perform an action that is neither on a correct path nor on a defined incorrect one. At this point, model tracing has nothing to say other than that it is incorrect. Model tracing systems may use repair theory (VanLehn 1983) to overcome the impasse, by backtracking and suggesting alternative actions which the student may adopt, until the trace is "unstuck". However, this is a non-trivial task since it is rarely clear where the repair should be made, and so the repairer may encounter a combinatorial explosion of potential paths (Self 1994).

DISCOVER (Ramadhan and Du Boulay 1993) maintains control of the model tracing process by providing two interfaces: a general one where students may construct solutions on their own without feedback, and a "guided phase" module, where they are restricted in what they can input. An alternative method is to build the student interface in such a way that only selected actions may occur. ELM-PE (Weber 1993) provides a syntax-based structure editor, which automatically fills in LISP statement slots with appropriate insertions, such that only valid LISP may be constructed.

In contrast, CBM tutors like SQL-Tutor intentionally place no such restrictions on the user—they are free to write their solutions in any order using whatever constructs they see fit. The solution is then evaluated *as a whole* according to whether or not it is syntactically correct and satisfies the semantics of the problem. The solution may therefore deviate radically from the correct solution, at which point the user's "intentions" are completely unknown. Some systems that suffer this problem try to

overcome it by forcing the student to make their intentions explicit. Bridge (Bonar and Cunningham 1988) breaks down the problem solving process into three steps. First, the student formulates their ideas in English. Then, they translate their informal ideas into plan specifications. Finally, they build the program code. Because it already has their intentions, Bridge is able to understand partially completed code. Similarly, Capra (Verdejo, Fernandez and Urretavizcaya 1993) breaks problem solving into three parts: problem extraction, relation to a class of solutions, and refinement to a final answer. Note however, that Capra does not allow the student to enter his or her own solution. Rather, they are "led" to one of a set of solutions stored in Capra's knowledge base. Such an approach has been criticised for making students dependant on being led to a solution, rather than developing their own problem-solving skills.

In the previous chapter, we developed a representation for constraints that makes the evaluation process transparent, in that a satisfied condition can be reversed, to show *why* it succeeded, while for a failed test we can see what the construct *should have been*. To satisfy hypothesis 1, it must be possible to build an algorithm that uses this representation to solve problems. We present hypothesis 2:

> **Hypothesis 2**: Using the representation defined in hypothesis 1, it is possible to develop an algorithm for solving problems and correcting student answers, which does not need further domain information to achieve this.

5.2 The approach

In a constraint-based model, each constraint can be thought of as a pair of conditions that reduce the solution space. The relevance condition represents a certain subset R of the solution space, which is the set of problem/solution states we are interested in. Similarly, the satisfaction state defines another subset S, which represents correct solutions given that our solution is a subset of R. This is depicted in Figure 4.

Each constraint divides the solution space U into four regions: (1) R-S, (2) S-R, (3) $R \cap S$, and (4) $U-(R \cup S)$. If the solution is in region 1, the constraint is relevant but not satisfied, i.e. it is violated. If the solution is in (2), it is not relevant so we are not interested in it. If it is in (3), it is both relevant and satisfied. A solution in 4 falls outside the scope of this constraint altogether. Therefore, only (1) is a problem; all

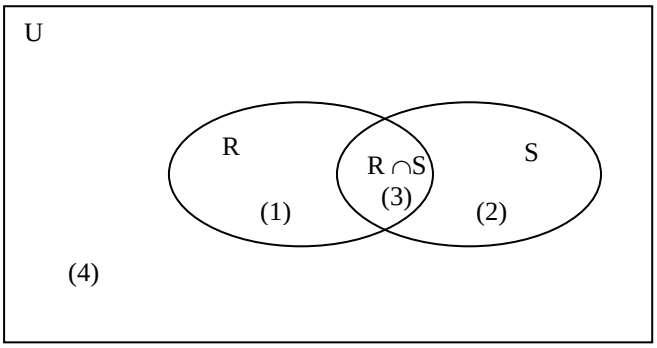

Figure 4. Solution Space

other regions do not signify a violation. To remove a violation we need to move the solution out of area (1) into *any* of the other regions.

Thus for each constraint we can either satisfy the constraint or render it no longer relevant. When this is true for *all* constraints, the solution is correct with respect to the constraints. If the constraint set is correct and sufficient, this will be a correct solution to the problem.

In essence this is a constraint satisfaction problem: the problem/solution must simultaneously satisfy (or not be relevant to) all constraints, and so is a difficult problem to solve. In practice we can define some heuristics that reduce this to an iterative problem, although the time taken to solve a given problem is not necessary linear. We describe these heuristics in the next section.

5.3 Problem solving with constraints

In the new representation, the (expanded) constraints make explicit all of the encoded domain knowledge: for any given constraint, all requirements of the ideal and student solutions are encapsulated in the MATCH and TEST pattern lists and the logical connectives between them. This means that the relevant constraints plus the variable bindings for each describes all that we know about the solution relevant to the problem, given our current domain knowledge.

Consequently, given a complete domain model, we can rebuild the solution from just the relevant constraints and their bindings. In the following example we list the match patterns resulting from the evaluation of a *correct* student solution. Only matches related to the student solution are listed, with variable bindings substituted

back in to give bound fragments of the student solution. The resulting list contains many fragments that subsume others, i.e. they are a more specific version of one or more other fragments. We have omitted the subsumed fragments for clarity.

Ideal Solution

```
SELECT    title
FROM      movie
WHERE     director=(select number from director
            where fname='Stanley' and lname='Kubrick')
```

Student Solution

```
SELECT    title
FROM      movie join director on director = director.number
WHERE     lname = 'Kubrick' and fname = 'Stanley'
```

Bound Matches (with subsumed fragments omitted)

```
SELECT (title ?*)
FROM   (?* movie JOIN director ON director = director.number ?*)
WHERE  (?* lname = 'Kubrick' and ?*) (?* = 'Kubrick' and fname ?*)
       (?* 'Kubrick' and fname = ?*) (?* and fname = 'Stanley' ?*)
```

The WHERE clause contains more than one fragment. These are “spliced” together by joining overlapping fragments and removing the duplication. Wildcards are then deleted, yielding:

```
SELECT    (title)
FROM      (movie JOIN director ON director = director.number)
WHERE     (lname = 'Kubrick' and fname = 'Stanley')
```

which is the same as the student solution.

5.4 Correcting an erroneous solution

To provide tailored feedback we produce a correct solution that is as close as possible to the student’s attempt based on pattern matches from the relevant constraints. The previous example illustrated that the constraints may contain sufficient information about a correct solution to rebuild it. In the case of an *incorrect* solution, the fragments obtained from satisfied constraints tells us about the correct parts of the solution, while violated constraints indicate parts of the solution that must be repaired. To build a correct solution from a mal-formed one, we correct each violated constraint and add the resulting fragments to those obtained from the satisfied constraints.

There are three types of constraint violation that may occur: a MATCH against the student solution fails; a MATCH against the ideal solution fails; and a

TEST/TEST_SYMBOL fails. The first failure type indicates that one or more terms are missing from the student solution. This is corrected by adding the fragment for the failed MATCH. The second failure indicates that there are one or more extraneous terms in the student solution, which can be corrected by deleting the corresponding match fragment. Finally, a failed TEST indicates that one or more variables in a previous match fragment are incorrect. This is corrected by substituting the *expected* value for those variables.

The correct solution is built by beginning with the set of solution fragments created by the satisfied constraints and passing it through a modified version of the constraint evaluator, which accepts bound matches (including wildcards) as input. Each time a constraint is violated, action is taken (as indicated previously) to remove the violation. The fragment set is then checked for subsumed fragments, which are removed. The cycle then repeats until no constraints are violated. At this stage the fragments are spliced together as illustrated in the previous section and wildcards removed, yielding a corrected solution.

5.5 Examples of solution correction

SQL-Tutor has been subjected to four prior evaluation studies (Mitrovic, Martin and Mayo 2002) where, as well as collecting general statistics about the performance of the system, we logged the students' attempts. There were many cases where the student solution was fundamentally different to the ideal solution, and so the feedback given for a partial solution was not relevant to their answer: Of all partial or full solutions presented to a student, 22 percent were either fundamentally different to the student solution or varied such that the student made unnecessary alterations to their answer. We now examine the performance of solution generation in two such situations. The first is a simple example to illustrate how the method works, while the second demonstrates its flexibility. As detailed in the previous section, generating the correct solution involves testing the solution against the constraints, extracting solution fragments from satisfied constraints, and adding, removing or modifying fragments for violated constraints. This process is repeated until all fragments are

valid and none are missing. The solution is then built by splicing together the remaining fragments.

The following example is taken from one of the evaluation logs. The student has made two mistakes. First, in the SELECT clause, they have used a "." instead of a "," to separate two fields. Second, they have used "=" instead of ">=" in the WHERE clause. The system first tests the solution against the entire constraint set to determine whether or not it is correct, and discovers that six constraints are violated:

```
Ideal solution:
   (SELECT (lname , fname))
   (FROM (director ?*))
   (WHERE (born >= 1920 ?*))

Student solution:
   (SELECT ((lname . fname)))
   (FROM (director ?*))
   (WHERE (born = 1920 ?*))

VIOLATED:(650 462 6500 1802  192 5)
```

The student has made the following errors:

- They have used "." instead of a comma to separate attribute names. To the system they have omitted attributes (constraint 650) and included a spurious one (constraint 6500). This attribute appears to come from an unnecessary table ("lname" – constraint 1802), which does not appear in the "FROM" clause (constraint 192), and is not a valid database name (constraint 5).
- They have used an incorrect comparison operator ("=" instead of ">=") to compare "born" to "1920" (constraint 462)

The system now tries to correct the semantics first, since there is no point in correcting the syntax of elements that are not actually required. Four semantic constraints are violated, the first being 650, which tests that all required attributes are present in the SELECT clause (see appendix D for the constraint definitions). The system does not recognise `lname` and `fname` in the student solution, because they have been interpreted as a single attribute, `lname.fname`. This fault is corrected by adding the two missing fragments, i.e. the two attribute names:

```
Student solution:
   (SELECT ((lname . fname)))
```

```
   (FROM (director ?*))
   (WHERE (born = 1920 ?*))

VIOLATED:(650 462 6500 1802)
ACTIONING 650
   ADDING FAILED MATCH ((SELECT fname ?*))
   ADDING FAILED MATCH ((SELECT lname ?*))

New solution:
   (SELECT ((lname . fname)) (fname ?*) (lname ?*))
   (FROM (director ?*))
   (WHERE (born = 1920 ?*))
```

The solution is now retested, and still fails three constraints, as shown below. The next one, 462, tests that the operator used to compare two terms is the right one. A TEST fails, so the incorrect value of "=" is changed to the correct one of ">=":

```
VIOLATED:(462 6500 1802)

ACTIONING 462
TEST IS FAIL-TEST

New solution:
   (SELECT ((lname . fname)) (lname ?*) (fname ?*))
   (FROM (director ?*))
   (WHERE (born >= 1920 ?*))
```

The semantics are again tested below, and there are still two constraints failing. 6500 tests that there are no extraneous attributes in the SELECT clause, by trying to match all attributes in the student solution to the ideal solution. This fails, so the offending extra attribute (`lname.fname`) is removed:

```
VIOLATED:(6500 1802)

ACTIONING 6500

New solution:
   (SELECT (lname ?*) (fname ?*))
   (FROM (director ?*))
   (WHERE (born >= 1920 ?*))
```

The solution is now semantically correct (removing `lname.fname` has corrected constraint 1802 as a side effect). The syntax is now checked, and found to be correct. The next step is to splice the fragments together. Only SELECT has more than one fragment and they do not overlap, so they are simply concatenated. The spliced solution is now tested (below) and found to fail the syntactic constraint 350, which

checks for commas separating attribute names by looking for the match `(?a1 "," ?a2)`. This fails, so the missing fragment is inserted:

```
VIOLATED:(350)
ACTIONING 350
   ADDING FAILED MATCH ((SELECT lname , fname))

New solution:
   (SELECT NIL (lname , fname))
   (FROM (director ?*))
   (WHERE (born >= 1920))
```

The solution is now correct. However, this is a fairly trivial example because the student and ideal solutions were very similar. Consider the next example, where the student uses a different strategy to the system:

```
Ideal solution:
  (SELECT "title")
  (FROM "movie")
  (WHERE "director=(select number from director where
         fname='Stanley' and lname='Kubrick')")


Student solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE (lname = 'kubrick and fname  = 'Stanley'))
```

Here, the student has used a JOIN, whereas the ideal solution uses a nested query. The only errors are that the trailing quote is missing off the string `'kubrick'`, and the first letter in this string should be uppercase. The first error means the system doesn't know where the string ends, so arbitrarily chooses the "=" as the terminator, giving a (malformed) string of `'kubrick and fname`. This causes nine constraints to fail. The system now tests the semantics, and tries to correct the first violation. Constraint 372 tests that all required strings are present, by trying to match all strings in the ideal solution to the student answer. This fails, so the missing fragment `'Kubrick'` is inserted:

```
VIOLATED:(372 239 2730 1514 999)
ACTIONING 372
   ADDING FAILED MATCH ((WHERE ?* 'Kubrick' ?*))

New solution:
```

```
   (SELECT (title ?*))
   (FROM   (movie join director on director = (director . number)))
   (WHERE  (lname = 'kubrick and fname  = 'Stanley')
           (?* 'Kubrick' ?*))
```

The semantics are retested, and constraint 2370 fails (below). This constraint tests that attributes are compared to the *correct* string. The failure was a TEST of a value, so it replaces the incorrect string value of "`'kubrick and fname`" with the correct one of `'Kubrick'`. The algorithm also eliminates subsumed fragments during processing, so the previously added fragment of `(?* 'Kubrick' ?*)` is deleted:

```
VIOLATED:(2730 1514 999)
ACTIONING 2730
TEST IS FAIL-TEST

New solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE (lname = 'Kubrick' = 'Stanley'))
```

At this stage the algorithm is satisfied with the semantics so the syntax is checked, and three constraints fail. The first is that there are two conditions without a logical connective:

```
VIOLATED:(347 454 4629)
ACTIONING 347
   ADDING FAILED MATCH ((WHERE lname = 'Kubrick' AND = 'Stanley'))

New solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE NIL (lname = 'Kubrick' AND = 'Stanley'))
```

Note that the choice of "AND" (rather than "OR") is arbitrary; if it is not correct another constraint should be violated that will substitute the correct value. Now, there is a malformed condition "`AND = 'Stanley'`", which fails a TEST that expected the term preceding the "=" to be an attribute. This is corrected by replacing AND with an arbitrary attribute:

```
VIOLATED:(454)
ACTIONING 454
TEST IS FAIL-TEST

New solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE (lname = 'Kubrick' number = 'Stanley' ?*))
```

The constraint just fixed (347—missing logical connective) now fails again, and is corrected again:

```
VIOLATED:(347 203 20_A)
ACTIONING 347
  ADDING FAILED MATCH ((WHERE lname = 'Kubrick' AND number =
                        'Stanley' ?*))
New solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE NIL (lname = 'Kubrick' AND number = 'Stanley' ?*))
```

The next two corrections illustrate how the greedy approach used can cause unnecessary work to be performed. The first corrects the fact that the newly added attribute (number) is ambiguous. Immediately following this, constraint 20_A determines that ‘number’ is of the wrong type (numeric) to be compared to a string, so swaps it for another attribute of the correct type. Thus, the results of the previous step are discarded.

```
VIOLATED:(20_A 203)

New Solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE (lname = 'Kubrick' AND (movie . number) = 'Stanley' ?*))

VIOLATED:(20_A)

New solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE (lname = 'Kubrick' AND address = 'Stanley' ?*))
```

The syntax is now correct, and the semantics are again checked. Constraint 175 (below) discovers that `address` is the wrong attribute being compared to `'Stanley'`, and substitutes the correct one:

```
VIOLATED:(175)
ACTIONING 175
TEST IS FAIL-TEST

New solution:
   (SELECT (title ?*))
   (FROM (movie join director on director = (director . number)))
   (WHERE (lname = 'Kubrick' AND fname = 'Stanley' ?*))
```

The solution is now correct. It is very similar to what was originally entered, and continues to use a different strategy to the ideal solution.

5.6 Discussion

In the second example we generated a corrected version of an incorrect student answer despite the student and ideal solutions being fundamentally different: the ideal solution used a nested SELECT, while the student solution included a JOIN. This suggests that the new representation is sufficient for the domain model to be generative. However, there is room for improving efficiency. First, the algorithm described is greedy in that it performs actions for all failed constraints and their bindings, even though they may be undone by actions for later constraints. In particular, it may generate new fragments that are later deleted again. This is inefficient, and we need to explore ways to reduce the amount of redundant work performed, without adding unnecessary complexity.

Second, each failed constraint results in just one action, based on the first item in the condition that failed. Some constraints are encoded as a MATCH, with either unrestricted or partially restricted variable terms, which are then further restricted by other tests. If the initial MATCH fails, it will generate an action that adds a fragment containing similarly general terms. These terms will remain until they are picked up by another constraint failure (possibly for this same constraint) in a subsequent processing step. It may be possible to perform more work at each action by incorporating more than just the first failed step in the condition, thereby reducing the number of iterations overall.

Finally, there is no guarantee that the solution being built will converge. Often, there are multiple ways to satisfy a failed constraint, some of which may lead to extraneous constructs being added to the solution. It is important that these are removed, and that the solution does not “oscillate” between two or more potential solutions (for example, by continually adding an incorrect new term, only to have it later removed because of another constraint violation). In the next section we describe how we have attempted to avoid such situations, however they remain possible.

5.7 The problem-solving algorithm

The problem solver tries to correct the student's answer by removing violations, one constraint at a time. It is greedy on three counts: (1) only the first constraint (in the order they were encoded) is initially selected for repair, (2) each constraint may be violated more than once, but only the first instance is initially selected for repair, and (3) when there are multiple ways to satisfy a constraint (i.e. an OR disjunct), only the LAST is selected. The latter is a simplification that obviates the need for the pattern matcher to retain information about previous failed bindings: at the time the constraint fails, the information about the last failed disjunct will be still available. This level of greed is probably far from optimal.

5.7.1 Algorithm overview

The problem-solving algorithm consists of two main parts: extensions to the pattern matcher to retain all the information needed about fragments and failures, and the set of routines that correct the errors. Further, the former can be further divided into collecting fragments, and building a *corrective action list*, for the error correction algorithm to carry out. Problem solving therefore consists of alternately testing the solution and gathering fragments and corrections, and correcting the first failed constraint. This continues until no constraints are violated, or the algorithm gives up. The latter may occur if the constraints cause a loop. The fragments resulting from the corrective actions are then spliced together into a single solution, which is again tested for fidelity with respect to the constraints. Finally, the solution is tidied up. The overall algorithm is given in Figure 5. Note that MAX-TOTAL-ATTEMPTS, MAX-CORRECTIONS and MAX-TIDIES are constants that determine how many times the algorithm should try to correct errors before giving up. Currently they are all set to 20. Each of the main components is now described.

5.7.2 Collecting corrections

Correction information is collected by the pattern-matcher because it is tied to the bindings that were valid up to the failure. The corrective action list is maintained separately to the binding list. When a MATCH fails, MATCH-BINDINGS inserts an entry into the corrective action list containing a tag indicating that the type of error was a match failure, whether the test was for the ideal solution or the student solution, and the set of bindings that were current at the time. This provides the full context needed to perform the correction. Similarly, TEST adds an entry for each failed test. The following is an example of a corrective action list entry:

```
(174
 (FAIL-TEST
   ((NIL (?A1)) ?A2                                          (1)
```

```
Correct Answer:

Test answer against the syntactic and semantic constraints, gathering
all fragments and corrective actions

If any violations
   Loop until no longer violated, or MAX-TOTAL-ATTEMPTS exceeded
      Loop until no more violations, or MAX-CORRECTIONS exceeded
         perform all constraint violation corrections

         test the solution against the semantic and syntactic
         constraints, gathering corrections

      Splice the resulting fragments into a complete solution

      test the solution against the semantic and syntactic
      constraints, gathering corrections

    Loop until no more violations or MAX-TIDIES exceeded
      test the solution against the tidying constraints, gathering
      corrections

      Perform corrections

    If any semantic or syntactic constraints still violated
         return FAIL
    else
         return corrected solution
else
    return the original solution
```

Figure 5. Solution generation algorithm

```
   ((?TEST SS ((((aanom integer)…)(?A2 integer))(?A2 integer))) (2)
    (?FRAG-R WHERE (?* ?N2 ?OP2 ?C ?*))                          (3)
    (?OP2 . =)                                                   (4)
    (?TEST SS ((NIL (?A2)) ?N2))                                 (5)
    (?A2 . number)                                               (6)
    (?N2 . number)                                               (7)
    (?TEST IS (((. 0 1 … 9) ?104_D1) ?104_D1)) (?104_D1 . 1)     (8)
    (?T1 . tape)                                                 (9)
    (?TEST IS ((NIL (?A1)) ?N1))                                (10)
    (?A1 . times)                                               (11)
    (?C . 10)                                                   (12)
    (?OP . >=)                                                  (13)
    (?N1 . times)                                               (14)
    (?TOP . TOP))                                               (15)
 )
)
```

The input that caused this failure was:

```
Ideal solution:  WHERE times = 10

Student solution: WHERE number = 10
```

This entry is for constraint 174, which has failed a test. Line (1) indicates the failed test: `?A2` failed to equate to `?A1`. Lines (2) through (15) are the binding set for this test, including fragment and test information. Line (3) is a fragment for a successful match. Lines 5, 8 and 10 record successful tests, while the rest give the actual values for the bindings, for example (4) indicates that the value of `?op2` is "`=`". Line (6) shows that the value of the variable `?A2` is "`number`", while (11) indicates that `?A1` is "`times`", hence the failure of the test for equivalence of `?a1` and `?a2`. The correction algorithm will correct the value of `?A2` to "times", however `?A2` does not appear in the fragment in (3), so will have no effect on this fragment. Entry (5) indicates that `?A2` and `?N2` are equivalent, so the correction algorithm will also update `?N2`, which does appear in the fragment. Hence, the fragment will be changed from "`number = 10`" to "`times = 10`", correcting the error.

5.7.3 Fixing errors

The constraints are divided into three sets: *semantic*, *syntactic*, and *tidying* constraints. Semantic constraints are those that compare the student solution to the ideal solution. Their purpose is to ensure that the student solution contains the necessary terms to

solve the problem. For example, the following semantic constraint checks that the student has provided all the necessary attributes for sorting:

```
(531

"Check whether you have specified all the necessary attributes in
the ORDER BY clause."

(and
    (match SS ORDER_BY (?what ?*))                              (1)
    (match IS ORDER_BY (?* (^attribute-p (?n ?a ?t)) ?*))       (2)
)

(match SS ORDER_BY (?* (^attribute-p (?n2 ?a ?t)) ?*))          (3)

"ORDER BY")
```

Condition (1) checks that the ORDER_BY clause of the student solution is not null. Then, (2) binds `?n` to all valid attribute names in the ORDER_BY clause of the ideal solution, and also binds `?a` to the attribute, and `?t` to the table name that are either implicit (in the case of an unqualified name) or explicit in each `?n`. The satisfaction condition (3) then tests that there can be bound a valid attribute name `?n2` for *each* (`?a ?t`) value pair, such that `?n2` represents the same attribute and table name. In other words, for each physical database attribute implied by a name in the ORDER BY clause in the ideal solution, there must also be some name in the student solution that represents the same physical attribute.

Syntactic constraints test that the student solution is valid SQL, with no reference to the problems being solved. For example, the following syntactic constraint tests that all names in the FROM clause are either valid table names or valid attribute names.

```
(146

"You have used some names in the FROM clause that are not from this
database."

(match SS FROM (?* (^name ?n) ?*))

(or-p
          (test SS (^attribute-p (?n ?a ?t)))
          (test SS (^table-in-db ?n)))
)
```

A side effect of the greedy approach to problem solving is that some changes may be made to the student solution that later turn out not to be needed and which degrade the quality of the solution. In SQL attributes may be either qualified (e.g. `SELECT movie.director`) or unqualified (`SELECT director`). Attributes must be qualified if they would otherwise be ambiguous. When correcting an error in SQL-Tutor where an attribute has been used for which no table exists in the FROM clause, the algorithm may add a new table to FROM, rather than remove the attribute, only to later remove both the new table and the offending attribute because they were superfluous. In the meantime however, the problem-solver may qualify one or more other attributes because the addition of the new table made them ambiguous. This is not an error since any attribute can be legally qualified, but it degrades the quality of the solution and may lead the student to think it needed to be qualified. To solve this dilemma and others like it, tidying constraints are used to effect *desirable* properties of the solution. In SQL-Tutor, over-qualification is the only such case. An example of a tidying constraint is given in Figure 6. This constraint tests that if there exists a table `?t1` in the FROM condition to which some attribute `?n` exists in the SELECT that is an attribute of this table, and there can be found no other different table `?t2` of which the attribute `?a` of `?n` is also an attribute, then `?n` need be the attribute name only, i.e. not the qualified name `(?t . ?a)`.

The main (semantic and syntactic) constraints are split into two sets for efficiency. The algorithm generally prefers adding new fragments or modifying existing terms (solution growth), to deleting fragments (solution pruning). This is because terms that are incorrect (such as an extraneous attribute name) may be part of a wider construct that is mostly correct. If the incorrect attribute were deleted, it may render the larger construct syntactically incorrect, causing it too to be deleted, and much of the student's original attempt will be lost. However, if the erroneous attribute is replaced by the correct one, the wider construct may now be correct.

```
(2
"You have qualified an attribute in SELECT that would not be
ambiguous without the qualification."
(and
   (match SS FROM (?* (^table-in-db ?t1) ?*))
   (match SS SELECT (?* (^attribute-of (?n ?a ?t1)) ?*))
   (not-p
      (and
           (match SS FROM (?* (^table-in-db ?t2) ?*))
           (not-p (test SS ((?t1) ?t2)))
           (test SS (^attribute-in-db (?a ?t2))))
      )
)

(test SS ((?a) ?n))

"SELECT")
```

Figure 6. Tidying constraint

A side effect of preferring solution growth to solution pruning is that the algorithm may go to considerable lengths to grow a new construct, only to discover that it was based on a partial construct that was unnecessary. To reduce the chance of this the solution is always checked semantically first, which will correct, add and, if necessary, prune as many incorrect terms as possible. Once the solution satisfies all of the semantic constraints, it is tested against the syntactic constraints to ensure that all the current constructs are syntactically correct.

At this stage more terms may have been added that are semantically incorrect. Consider the following constraint:

```
(455
"You need to specify an attribute to compare the string constant to
in HAVING."

(match SS HAVING (?* (^rel-p ?op) (^sql-stringp ?s) ?*))

(match SS HAVING (?* (^attribute-p (?a ?att ?table)) ?op ?s ?*))

"HAVING")
```

This constraint tests that a relational operator and a string (e.g. “`= 'Ferrari'`”) is preceded by *any* valid attribute. If this constraint is violated, it will add an attribute into the HAVING clause. The semantic constraints will then need to ensure that it is the *correct* attribute. The problem solver thus loops, alternately testing the semantics then the syntax, until all constraints are satisfied. Finally, the tidying constraints are

checked. These may be applied independently of the other constraints because they are guaranteed not to make any changes that will violate the main constraints.

Only one constraint is corrected after each test of the constraints. This is necessary because the satisfaction condition of many constraints contains more than one test. If only the first test failure were corrected followed by the corrective action for another constraint, the second could cause the rest of the first constraint never to happen, causing looping or extraneous fragments. Consider the following two constraints:

```
(1
"Check you are comparing the correct attribute to the nested select
in HAVING"
(and
   (match SS HAVING (?*w1 (^attr-name (?n1 ?a1 ?t1)) (^rel-p ?op)
       ("SELECT ?*w3 "FROM" ?*w4 (^table-in-db ?t2) ?*w5))
   (match IS HAVING (?* ?agg "(" ?what ")" ?op2 "(" SELECT ?*
       "FROM" ?* (^table-in-db ?t4) ?*))
)
(and
  (match SS HAVING (?*w1 ?agg "(" ?what ")" ?op "(" SELECT ?*w3
       "FROM" ?*w4 ?t2 ?*w5))
  (not-p (match SS HAVING (?*w1 ?n1 ?op "(" SELECT ?*w3 "FROM" ?*w4
       ?t2 ?*w5)))
)
"HAVING")

(2
"Check the relational operator you are using in the HAVING clause."
(and
  (match IS HAVING (?* ?agg "(" ?what ")" ?op1
       "(" "SELECT" ?* "FROM" ?*  ")" ?*))
  (match SS HAVING (?* ?agg "(" ?what ")" ?op2
       "(" "SELECT" ?* "FROM" ?*  ")" ?*))
)

(test SS ((?op1) ?op2))

"HAVING")

SS: aawon > ( select aawon from…)
IS: avg(aawon) >= (select aawon from)
```

Constraint 1 ensures that an aggregate function is used rather than just an attribute, if this is present in the ideal solution. Constraint 2 tests that when an aggregate function is compared to a nested query, the correct relational operator is used. In the example, both constraints are violated. Constraint 1 first adds a new match (in bold), giving:

```
aawon > ( select aawon from…)
avg(aawon) > ( select aawon from…)
```

Suppose we now go on to correct the violation for constraint 2. In this case a test has failed, so we substitute the correct value for the operator:

```
aawon > ( select aawon from…)
avg(aawon) >= (select aawon from)
```

We now retest the constraint set again. Constraint 1 once again fails, giving:

```
aawon > ( select aawon from…)
avg(aawon) >= (select aawon from)
avg(aawon) > ( select aawon from…)
```

Finally, on testing the constraint set one more time, (2) fails again, and corrects the added fragment, which is now deleted because it is a duplicate, and we are back to where we started:

```
avg(aawon) > (select aawon from)
avg(aawon) >= ( select aawon from…)
```

The algorithm is now looping. If the constraints were corrected one at a time and retested, the following would happen instead: First, constraint 1 fails, as previously:

```
aawon > ( select aawon from…)
avg(aawon) > ( select aawon from…)
```

The constraint set is tested again. Constraint 1 fails again, and removes the extra fragment:

```
avg(aawon) > ( select aawon from…)
```

Finally, constraint 2 fails, and corrects the operator.

```
avg(aawon) >= ( select aawon from…)
```

Each failed constraint is therefore actioned and retested repeatedly until it is no longer violated. This ensures that another constraint does not become relevant and cause looping or extraneous fragments.

5.7.4 Putting it all together

On completion of the correction process, the solution now consists of a set of SQL fragments that must be combined into a single SQL statement. The individual fragments are concatenated, or *spliced* together, being mindful of two conditions: that

two or more fragments may represent different parts of the same SQL construct, and that the order of the various parts of the solution (e.g. conditions in a WHERE clause) should be the same as the student's solution as far as possible. The first is achieved by comparing each fragment with each other, to see if they overlap. If so, the overlapping portion of one fragment is removed before concatenation. Ordering is kept consistent by pre-sorting the fragments according to the student solution. Each fragment is compared to the original student attempt, and given a rank of at which input term they first (at least partially) match. For example, if a fragment has the same first term as the third term in the student input, it will be given a ranking of 3. In the case of a tie, the offending fragments are rechecked to see which continues to match in that position if more terms in the fragment are considered. The fragments are then sorted based on the ranking given.

The act of splicing the fragments can cause further constraint violations. For example, the two condition fragments "`lname = 'Kubrick'`", and "`fname = 'Stanley'`" will splice to form "`lname = 'Kubrick' fname = 'Stanley'`" which is incorrect, because there is no logical operator conjoining the two conditions. The spliced solution is therefore re-checked against the entire constraint set and necessary corrections made. This process iterates until the newly spliced solution no longer violates any constraints. In practice, re-correction tends to occur at most once.

5.8 Robustness testing

The solution generation method described is feasible but potentially unworkable. As stated earlier, one of the chief advantages of CBM over model tracing is that the constraint set need not be complete or perfect, because each constraint is used in isolation without chaining. This means that the effect of an error is highly localised. With solution generation the effect of errors in the constraint set is more severe. There are two problems that may arise, both of which can be catastrophic: that a valid construct is disallowed, and that an invalid construct is permitted. Disallowing valid constructs can cause looping, because the construct in question may be added by one constraint, only to be (erroneously) deleted by another. Allowing erroneous constructs

may cause the finished solution to contain spurious elements that were either produced by the student or worse, added by the algorithm.

Incorrectly encoded constraints may also cause serious problems, including looping. Consider the following constraint:

```
(1
"You need a logical operator between conditions in WHERE"

(match SS WHERE (?* ?w (^rel-p ?op) ?c
                    ?w2 (^rel-p ?op2) ?c2 ?*))

(match SS WHERE (?* ?w ?op ?c (("and" "or" ?conn) ?w2 ?op2 ?c2 ?*))

"WHERE")
```

This constraint is trying to ensure that all conditions are joined by a logical connective. However, it is encoded such that both the correct and incorrect versions of the condition pair would be accepted, e.g.

```
"fname = 'Stanley' and lname = 'Kubrick' and fname = 'Stanley'
lname = 'Kubrick'"
```

If another constraint modifies either the incorrect or the corrected pair, looping may result.

Solution generation therefore imposes a burden of correctness upon the constraint set: within the space of solutions to the problem set and *potential student solutions to the problem set*, the constraint set must be complete and correct. The former is a definable set that can be readily tested. The latter is impossible to define and potentially infinite. It is therefore impossible to ever say with certainty that the algorithm will always provide a correct solution based on the student's input: the best that can be said is that we are reasonably confident that a solution will prevail *n*% of the time, and that the algorithm will always terminate. The second claim that the algorithm will terminate is achieved by coding a halting condition, i.e. that the algorithm stops after a fixed number of attempts. The first claim that the solution should be correct *n*% of the time may only be empirically measured.

5.8.1 Testing robustness

The default constraint set for solution generation was a direct translation of the existing constraints in SQL-Tutor. It was then tested to ensure it could solve all

problems in the problem space, by presenting the algorithm with a blank log file and requiring it to generate the correct solution to each problem. This step resulted in many corrections to the constraint set. Some of these were simply existing coding errors. However, a large number were additions to the constraint set or modifications to existing constraints because the constraint set was *too loose*, and so would miss real errors. This is the trade-off for allowing an incomplete constraint set that CBM must live with. It also highlights a positive side effect of the solution generation algorithm: it can serve as a fairly rigorous means of testing the constraints.

The algorithm, together with the modified constraint set, was then tested by attempting to correct wrong answers submitted by students in two previous evaluation studies of SQL-Tutor. In each case 30 logs were chosen from the study and arbitrarily split into two groups of fifteen students. For the first study, the 30 logs were those with the most submissions (out of 46). For the second study, this was the entire set of logs. Both studies were voluntary, and the participants were all students from a university database course. They had attended several lectures on SQL prior to the study.

Table 1 lists the number of student attempts that were corrected for the first set of 15 logs. It also lists the proportion of attempts that fell into each of the following categories:

- **Not resolved**: the algorithm was forced to abort because it was looping;
- **Incorrect**: the algorithm terminated but the generated solution contained errors, e.g. extraneous constructs in the generated solution;
- **Strategy difference(s)**: the solution generated is correct but is an example of a completely different problem strategy, e.g. a JOIN used instead of a nested query;
- **Structural differences**: the solution is an example of the same strategy but contains significant differences, e.g. the argument of an aggregate function has been unnecessarily modified;
- **Minor differences**: the solution is correct and largely the same as the students, but contains some unnecessary minor differences, e.g. an attribute which was previously not qualified was unnecessarily qualified;

Log #	Problem Attempts	Not Resolved	Incorrect	Strategy	Structural	Minor	All OK
1	11	0	0	0	0	7	4
2	26	0	8	1	0	0	17
3	4	0	0	0	0	0	4
4	16	0	0	0	0	2	14
5	41	0	10	0	8	10	13
6	48	0	8	0	2	5	33
7	3	0	0	0	0	0	3
8	36	0	18	1	4	2	16
9	39	0	10	0	0	9	23
10	8	0	0	0	0	0	8
11	18	0	7	0	3	1	8
12	39	0	10	0	11	8	11
13	49	0	7	0	0	6	36
14	23	0	8	0	2	3	12
15	52	0	10	0	5	4	35
Total	**413**	**0**	**96**	**2**	**35**	**57**	**237**
%		**0**	**23**	**0.5**	**8**	**14**	**57**

Table 1. Results for the training set

- **All OK**: the solution is totally correct, differing from the student solution only where necessary.

For the training set, 71% of the attempts were satisfactorily corrected in that any unnecessary differences between the new solution and the student's attempt were minor, and in the majority of cases there were no such differences. Of the rest, 8.5% were correctly generated but had unnecessary differences that might confuse the student, while 23% were wrong. This last category is the most critical: the algorithm should seek to avoid ever presenting an incorrect solution to the student.

These problems were corrected by further modifications to the constraint set, until corrections to all student attempts fell in the "All OK" category. At this stage the algorithm can be shown to produce excellent results on a known dataset, but its performance on future student input is unknown. To gauge this we now tested the algorithm on a further set of 15 logs from the same student population. The results are summarised in Table 2. For this test set, nearly 96% of the attempts were satisfactorily corrected with almost all of these being completely correct. Of the rest, just 0.9% failed to terminate, with 2.3% resolving, but having errors in the solution. Three

Log #	Problem Attempts	Not Resolved	Incorrect	Strategy	Structure	Minor	All OK
16	9	0	0	0	0	0	9
17	21	0	0	0	0	1	20
18	1	0	0	0	0	0	1
19	10	0	1	0	0	0	9
20	52	0	4	0	0	0	48
21	5	0	0	0	0	0	5
22	3	0	0	0	0	0	3
23	3	0	0	0	0	0	3
24	32	1	0	0	0	0	31
25	22	2	0	0	0	1	19
26	14	0	0	0	0	0	14
27	22	0	0	0	0	0	22
28	18	0	2	0	0	2	14
29	39	0	0	0	3	0	36
30	60	0	0	0	0	0	60
Total	**311**	**3**	**7**	**0**	**3**	**4**	**294**
%		**0.9**	**2.3**	**0**	**1**	**1.3**	**94.5**

Table 2. Results for the first test set (same population)

problem attempts had a difference that was considered more than minor. Some of the errors were:

Failed to resolve:

- Failed to resolve when the student used a different numeric constant (e.g. “0.1 * rentals” instead of “rentals/10”)—caused by missing constraints;
- Difficulties when a numeric calculation was entered using an unexpected representation, such as extra parentheses—caused by missing constraints;

Wrong:

- Combination of both “*” and a list of all attributes in the SELECT clause—caused by missing constraints;
- Both an unaliased and an aliassed representation of the same attribute in the SELECT clause—missing constraints;
- “type = (comedy or drama)” instead of “type = ‘comedy’ or type = ‘drama’” lead to incorrect structures—missing constraints;

Structural difference:

- “not (critics = nr)” is missing the quotes around ‘nr’, but instead of inserting the quotes, the algorithm changed the condition completely to “critics != ‘nr’”—caused by the way the constraints were encoded: the nr is interpreted as an (erroneous) attribute rather than a mal-formed string;

Minor differences:

- ordering of SELECT clause attributes—caused by incorrect attributes being deleted, and their correct counterparts being inserted in a different position in the clause.

The above problems were then fixed where possible, and a further 15 logs tested from a different evaluation, representing a completely separate population. Of this set, a larger proportion of solutions were unacceptable compared with the previous set (9.5% compared with 3.2%). However, this was much better than the first set tested from the previous population. Only four constraints required correcting to produce no errors for this set.

Finally, another 15 logs were tested from this second evaluation study, giving 92% correct solutions. Recall that the motivation for this research was to try to reduce the likelihood that students would be shown a partial or full solution that was inconsistent with their attempt, or contained unnecessary changes. In this final test group, 22 ideal solutions were presented to students in whole or in part, of which seven (32%) differed sufficiently from their attempt that the student made unnecessary changes. After applying solution generation, only one of the presented solutions (less than 5%) differed unnecessarily from the student solution.

5.9 Conclusions

The solution generation algorithm successfully solved up to 95% of incorrect student solutions. Once trained on a set of 15 student logs, it was able to achieve a 95% rating on a further 15 previously unseen logs. After correcting the constraints to eliminate the failures for this set it satisfactorily corrected over 90% of student errors for a different population. Although this is a higher failure rate than for the previous test, the number of *constraints* requiring correction trended steadily downwards. Figure 7

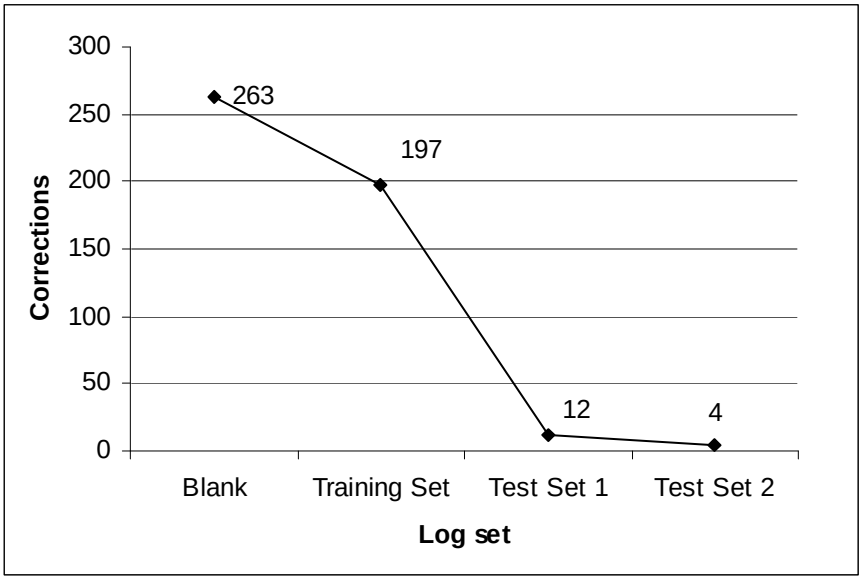

Figure 7. Constraints corrected per log

plots the number of constraints corrected or modified in response to problems with each group the algorithm was tested on. This graph shows that the number of modifications to the constraints decreased during the testing process. Further, more than half of the modifications were performed before any testing on live data was required, and a total of 96.6% of the modifications had been made to the knowledge base after testing just one set of 15 logs. This suggests that the behaviour of the algorithm is reasonably stable, in that once errors are eliminated it is unlikely that the system will fare significantly worse on subsequent populations.

Finally, we previously mentioned that correcting errors in the constraint set is desirable, because it leads to better diagnosis. We measured the level of misdiagnosis in the system prior to using solution generation, compared to after bugs had been fixed. For the test set of the second evaluation study (i.e. the *last* group tested above), the original version of SQL-Tutor misdiagnosed 16 cases out of 347 submissions (4.6%). In the final version (i.e. after correcting errors for all but this last group) there was only one error (0.3%), caused by the student using real attribute names as aliasses, which confused the constraint that checks that the correct attributes have been retrieved. Thus although correcting misdiagnosis was not a conscious goal, the system's diagnostic ability has improved as a result of the testing performed for solution generation, because it identified errors in the initial constraint set.

The robustness testing described indicates that the approach is promising and realistic on real, complex domains. Further refinement of the algorithm may further improve the situation. For example, more sophisticated ways of correcting errors than the very greedy approach described may reduce the likelihood of looping. We have therefore satisfied hypotheses 1 and 2, by showing that it is possible to develop a representation and algorithm that allows problems to be solved without the need for further domain information. That it works for a complex domain like SQL shows promise that it will be applicable to a wide range of domains.

6 Problem generation

The student model in SQL-Tutor is an overlay of the domain model consisting of a tally of how many times each constraint has been satisfied or violated. This is currently used to select the next problem, by choosing one where a target constraint is relevant. The target is the constraint that has been most violated. In this section we propose an alternative: generating a new problem tailored to the current situation.

Most ITSs do not generate their own problems, but choose from an enumerated set. However, some exceptions do exist. The Demonstr8 authoring tool (Blessing 1997) facilitates automatic problem generation but the domain (arithmetic) is very simple: the system merely selects random numbers from predefined ranges. XAIDA (Hsieh, Halff and Redfield 1999) is an example of problem generation in the more complex domain of device maintenance training. It generates four types of instruction: physical characteristics of a device, theory of operation, operating and maintenance procedures and troubleshooting. Each is supported by a separate "transaction shell", which is tailored to the particular type of instruction. In the "Physical Characteristics" section, XAIDA randomly selects pairs of attributes for the device currently being examined and then (again randomly) chooses a question schema that fits the characteristics of the attributes chosen. Such attributes may be parts of the device, values related to parts (e.g. volume of storage tank) etc. Information about the device and its parts are stored in a semantic network. Exercises for "Theory of operation" are similarly derived from a knowledge base about the device being learned. In this case, the knowledge base contains causal rules relating the state of certain components to the corresponding state of others. The author then generates a set of "cases"—combinations of device attributes and values that are instructionally useful. The system derives the "actions" (i.e. all of the consequences for other parts/attributes of

the device) based on the causal model. However, this is not really automatic problem generation since the “cases” define the structure of each problem. The “Procedures” section requires the author to model entire procedures, which XAIDA then randomly quizzes the user on. “Troubleshooting” is represented internally by a “fault tree”, which is randomly instantiated to depict a particular fault.

The systems described above perform some form of problem generation; however the problem they create is not *structured*. That is, they are limited to *selecting* a combination of values, instructional schemes etc, perhaps inferring some details of the *solution* from the underlying model (e.g. the “actions” in the XAIDA “theory of operation” module). In our case, we wish to take problem generation one step further: to generate a complex, structured problem (i.e. an SQL statement) without any problem-specific information being provided by the author. A comparable example from XAIDA would be if it could generate the valid “cases” for the theory of instruction based on the underlying model of device operation. Systems that use some sort of template to define the structure of the problem, such as XAIDA, run the same risk as manually authored problems: that the problem (or template) set is too small. Animalwatch (Arroyo, Beck, Beal and Woolf 2000), a system for teaching mathematics via word problems, similarly uses templates to generate new exercises, where the system simply instantiates numbers to create a new problem. Although Animalwatch contains 600 templates, students still complained of receiving the same problem twice but with different numbers.

SINT (Mitrovic 1996) is possibly a close comparison. This system teaches symbolic integration, using the student’s current behaviour to target a particular integration operator that the student has not learned. It then tries to construct a suitable example by inductive learning (Michalski 1983). This involves generalising the current exercise by climbing a directed graph of operators, where edges model dependencies, until an operator is reached for which all dependent others are still not learned. The tree is then descended, selecting appropriate operators and initialising constants, until a complete problem is built. The major difference is that the constraints in a CBM model are not related to each other explicitly. Rather, they are related implicitly in that making constraint C1 relevant may also render some other constraint C2 also relevant. For example, in SQL if constraint C1 requires that the

solution have a WHERE condition (because the data being selected needs to be constrained in some way), the constraints concerning the syntax of such conditions now also come into relevance, and hence they are implicitly related to C1. The problem solver must use these implicit connections to build a new solution, and hence a new problem. We explore this possibility and propose the following hypothesis:

> **Hypothesis 3**: CBM can also be used to generate new problems that fit the student's current model, and this is superior to selecting one from a pre-defined list.

6.1 Motivation

In SQL-Tutor there is no guarantee that an unsolved problem exists that matches the target. To overcome this, we propose generating a new problem using a procedure similar to problem solving. The steps are: (1) identify the target constraints; (2) produce a set of solution fragments from the relevance condition of the target constraints; (3) pass the fragments through the problem solver, generating a complete, novel solution; (4) convert the solution into natural language for presentation to the student. These steps are now described.

6.2 Identifying the target constraint

Previously, a single constraint has been chosen as the target. However, constraints are highly specific: in many cases a single concept will span multiple constraints. We have developed a method of automatically identifying the set of suitable target constraints from the student model using machine learning (Martin and Mitrovic 2000b). This algorithm uses the student model to classify all constraints as "learned", "not learned" or "unknown", based on the recent history of their application. A modified version of the PRISM machine learning algorithm (Cendrowska 1988) is then applied, which induces "rules" for the first two sets based on the text of the feedback message attached to each constraint. The rules induced for "not learned" describe the target constraints. Note that the target set may now also include constraints that have not been relevant yet, but which (according to their feedback

messages) are conceptually similar to failed constraints and so are unlikely to be known. This gives us a set of valid targets for any concept from which we may choose one or more as the basis for the new problem, allowing greater variability in problems we might generate. This algorithm is now described.

6.2.1 Motivation

In SQL-Tutor the student model is an overlay of the domain model. Each constraint has three counters: the number of times the constraint was relevant for the student solution, the number of times it was relevant for the ideal solution, and the number of times it has been violated by the student. These scores are used to select the next problem to present. The system currently chooses the constraint that has been violated most often, and picks an unsolved problem for which this constraint is relevant. This is adequate for problem selection but is constrained by the low-level nature of the individual constraints. We propose using machine learning to induce higher-level groups of constraints, which can add power and flexibility to the student model.

6.2.2 Increasing the knowledge depth

There is no point in adding information to a student model if it cannot be used to further guide the pedagogical process (Self 1990). The desire to add knowledge depth to the constraint-based model is motivated by the following:

- **To improve the selection of the next problem to present**. SQL-Tutor has only the individual violated constraints available to make this choice, and so can only present new problems if they use the actual target constraint. This artificially limits the pool of potential problems at each step;
- **To help the teacher understand the student's progress**. Constraints are such a specific representation of the problem domain that it is difficult for a human to gain an overall understanding of the student's competency or progress. A higher-level description of the areas of difficulty might be helpful to both teacher and student;
- **To aid feedback**. A system that can determine the concept behind an error can provide help about that concept, not just the particular instance at hand.

To this end, we propose that the inclusion of high-level concepts will increase the power of the constraint-based model to guide the pedagogical process.

6.2.3 Manually adding the concept hierarchy

We evaluated the possibility of adding a constraint hierarchy (Martin 1999) by creating one by hand for the set of semantic constraints in SQL-Tutor. Recall that semantic constraints compare the student's answer to an ideal solution to ensure that they have satisfied the question. An example of a semantic constraint is:

```
(p36
     "You need to order the resulting tuples - specify the ORDER BY
  clause"
      (match IS ORDER-BY (?what1 ?*))
      (match SS ORDER-BY (?what2 ?*))
  "ORDER BY")
```

This constraint checks that if the ideal solution has used an ORDER BY clause to sort the result, the student solution must do the same. To create a concept hierarchy the constraints were grouped into basic concepts of the SQL query language. These sets were then repeatedly partitioned into groups of constraints that share common sub-concepts, producing a tree with individual constraints as leaf nodes. The highest-level nodes in the tree (apart from the root) represent fundamental principles of SQL queries, such as "all tables present", "use of negation" and "sorting". Figure 8 illustrates the portion of the hierarchy that represents "all tables present," encompassing all constraints that check that the appropriate database tables have been referenced in the answer.

We formatively tested the proposed method by analysing the logs of students from an evaluation study of SQL-Tutor, and observing how they related to the proposed hierarchy. The participants were all volunteers, and had attended several lectures on SQL in a university database course. We observed from each log the set of problems the student attempted, and determined which constraints were relevant for each problem, and which were violated. A "hit list" was built up for each constraint, where a "✓" indicates the constraint was satisfied, and a "✗" indicates that it was violated.

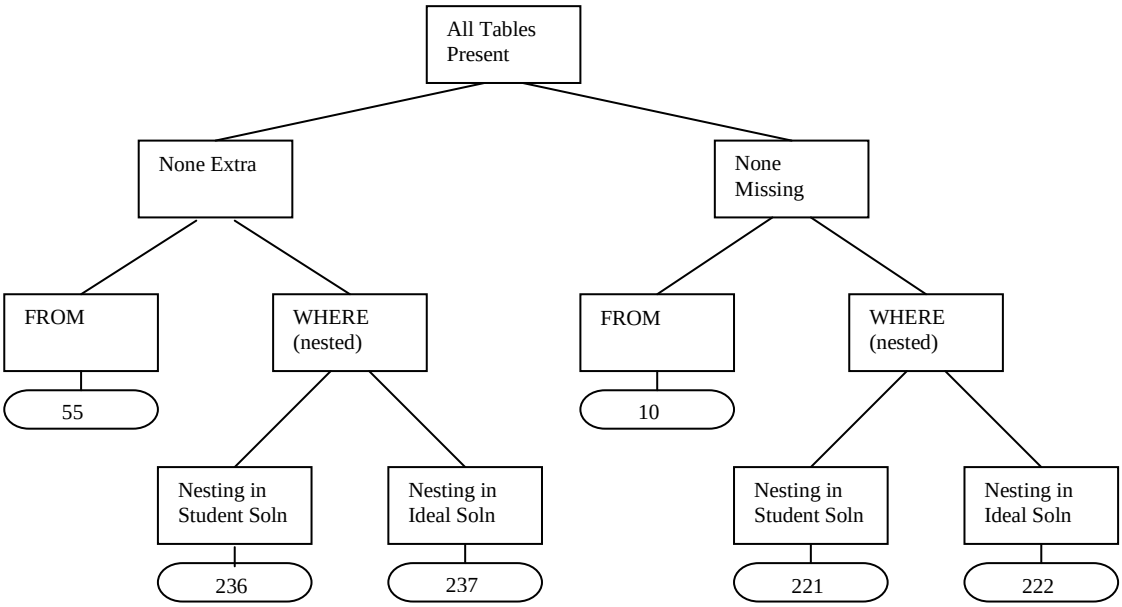

Figure 8. Concept hierarchy for "all tables present"

To allow for temporal variations in the student's knowledge, each constraint was classified as "LEARNED" or "NOT LEARNED" based on just the last four "hits" as follows:

- Any pattern containing "✓✓" indicates that the concept has been LEARNED;
- A constraint with only a single "✓" is (tentatively) considered to be LEARNED;
- A constraint with no hits is not labelled;
- Any other pattern indicates that the constraint is NOT LEARNED.

These heuristics were obtained by analysing the failure patterns of a population of students from their logs. We observed that the probability of satisfying a constraint rises exponentially with the number of previous successes, and after two consecutive successes the probability of satisfaction is nearly 100%. We assume that any subsequent failures after that are "slips".

The semantic constraint hierarchy was then pruned for each student so that the resulting hierarchy represents the concepts the student has failed to learn as generally as possible. Pruning was carried out as follows:

- Classify each constraint as described above;

- For each constraint classified as “NOT LEARNED”, ascend the tree towards the root until a node is reached where there are one or more nodes below it that are classified as “LEARNED”;
- Backtrack to the node below. This is the most general node that describes the concept that was not learned. Label this node “NOT LEARNED” and discard all nodes below it;
- Continue for all other constraints labelled “NOT LEARNED” that have not yet been discarded;
- Discard all nodes that have not been labelled “NOT LEARNED”, and do not have any nodes labelled as such below them.

This procedure was carried out for three students in the study who had solved at least 15 problems each, and who had failed many, average and few constraints. The resulting pruned hierarchies were collapsed into categories, where each is described by the labels of all nodes from the root down to each leaf. The results were as follows:

Student A failed many constraints, over a wide range of concepts, and appeared to still have much to learn about SQL. The pruned hierarchy contained the following categories:

- Sorting
- Aggregate functions
- Grouping
- NULLS - attributes

Student B failed fewer constraints than A, and attempted harder problems. Their pruned hierarchy contained just two highly specific categories:

- All tables present – none missing – nested SELECTs
- Negation – correct attributes

Student C fell somewhere between the other two, in that while their list of unlearned areas is longer than that of student A, more of them are very specific. For example, whereas student A still needed to learn “grouping” in general, student C was only

having problems with two smaller sub-areas. They attempted more questions, and more difficult ones, but failed quite a large number of constraints. Their categories were:

- Sorting
- Negation – correct attributes
- Grouping – attributes
- Grouping – existence – HAVING clause
- Expressions – arithmetic
- Expressions – non-arithmetic – DISTINCT
- All tables – none missing – nested SELECTs

We found that the results from the hierarchy appeared to be a good representation of the areas the students demonstrably had problems with. For the advanced student the hierarchy returned a small number of highly specific descriptions corresponding to a low number of constraints that were yet to be learned. For the less advanced student the hierarchy returned a set of very general descriptions (such as “grouping” and “sorting”), representing large numbers of potentially unlearned constraints. For the moderate student a larger set of reasonably specific descriptions were returned, indicating a good basic understanding of SQL but still quite a few specific areas to be mastered. Note that the hierarchy tells us nothing about what the student *does* know. For example, student B did not attempt any sorting problems, so the absence of any categories relating to sorting does not imply that it is learned. It would be possible to build another collapsed tree that represented the *learned* concepts in the same way, but this is dangerous. For example, if a student has correctly used just one SORTING constraint out of 11, it is not correct to say they have learned the concept of sorting, whereas if they have only encountered that same single constraint and violated it, it seems reasonable to assume they need to learn more about that concept.

It may seem odd that student C finished (after 25 problems) with so many discrete areas that they were having difficulty with, which suggests they moved on from concepts they were struggling with, or that they were attempting problems involving many concepts that they had not yet learned. In fact, both of these were true, for two

reasons. First, SQL-Tutor contains a fairly limited problem set—the largest problem set for a single database contained just 38 problems. Second, problems were selected according to the currently most violated constraint. This strategy can easily fail because there may not be another problem for which the same constraint is relevant. This is exacerbated by the specificity of the constraints. If a problem involving a *similar* constraint could be used, the system would more successfully focus on the target concept . This highlights the need for a more general view of the constraint set. Also, this approach gives no control over the number of new concepts introduced. Therefore, although the student is currently focussing on one aspect, such as sorting, the system might inadvertently introduce another new concept, such as grouping.

6.2.4 Inducing the student model using machine learning

The results for the hand-coded hierarchy were encouraging. However, there are two disadvantages. First, the hierarchy, like the constraint set, must be maintained. Any new constraints that are not added to the hierarchy will not be visible to processes that rely on it. If more than one person maintains the system, it is probable that new constraints could "slip through the cracks." Second, the hierarchy represents just one way of looking at the constraints: there may be others that are equally valid. More importantly, the same hierarchy may not fit all students. For example, the structure used in (Martin 1999) is heavily based around functional features such as "sorting", "grouping", "tables", "expressions" etc. Details such as nested queries or name aliasing are "hidden" lower down in the hierarchy. However, a particular student may have mastered the basics of SQL but repeatedly have problems with nesting queries. The relevant constraints for this type of problem are scattered throughout the hierarchy. Using Machine Learning would overcome both of these difficulties, making the student model more flexible and easier to maintain.

As described earlier, the hand-coded constraint set was produced by repeated partitioning of the constraint set based on key concepts such as "grouping." These concepts were determined by examining each constraint to identify its main function. However, the constraints already contain a description of what they do: the feedback message attached to each constraint is a concise description of the underlying concept being tested. We therefore propose that we can use Machine Learning to induce a

hierarchy from the basic student model using the feedback messages as input. This is an example of student model induction from multiple behaviours (Sison and Shimura 1998).

We analysed the text of the feedback messages for the set of semantic constraints and determined which words were likely to be keywords. In practice we kept all words except those that were highly likely to be superfluous, such as “a”, “the”, “and” etc. Some parsing of the messages was also necessary to remove suffixes. The resulting set of words formed the set of attributes, where each attribute has a value of “present” or “not present.” The set of “Not Learned” constraints was then converted into the set of positive examples, with attribute values determined according to which words were present. Similarly, the “learned” constraints formed the negative example set. These two sets were then combined to produce a training set.

We then induced modular rules for the class “Not Learned” using a similar algorithm to PRISM (Cendrowska 1988), except we only considered the attribute-value pairs with value “present”, because the absence of a word does not necessarily imply that it is not relevant. Each candidate attribute was given a score based on simple probability, i.e.

$$\text{Score} = \frac{p}{p+n} \tag{3}$$

where p is the number of positive examples where this attribute has the value "present", and n is the number of negative examples for this attribute value.

The set was then partitioned according to the attribute with the highest probability and coverage. If the probability is less than unity, those instances with a 0 for this attribute are removed, and the process repeated until unity is obtained, and the rule is now fully induced. All instances covered by the rule are now removed and the probability score for the remaining attributes is then recalculated, and the next rule induced. The process is repeated until no positive instances remain.

The resulting rule set describes the “Not Learned” constraints and is used to classify the constraints that have not yet been used by the student. If a constraint satisfies one or more rules, it is likely the student has not learned the underlying concept yet. For example, from the induced rules Student A is unlikely to have

learned any constraint that contains any of the following terms in the feedback message:

- “ORDER BY”, or;
- “function”, or;
- “GROUP BY”, or;
- “Grouping”, or;
- “NULL”, and “condition”.

6.2.5 Evaluation

If the rules we have induced represent concepts that a student has not learned, we expect each student’s “Not Learned” constraints to be grouped together such that those constraints that are described by each rule are related and are unlikely to have been learned by the student, given their observed behaviour. In the case of the hard-coded hierarchy, this appeared to be the case. We therefore used the results from (Martin 1999) as a benchmark for this evaluation.

We produced rules for the same three students as were used in (Martin 1999), and used them to classify the remaining unused constraints. We then compared the results. Table 3 illustrates the results obtained. “Induced rule” shows the rule induced using the Machine Learning method, compared to “Hierarchy category”, which is the most similar node from the hard-coded hierarchy, in terms of the name of the category and the constraints it represents. “Correctly classified” indicates the number of constraints that were not labelled (i.e. they had not been relevant) that the induced rule included, which were the same as constraints in the hard-coded category. Note that a “0” in this column indicates that neither the induced rule nor the hierarchy category generalised beyond the constraints the student had violated. “Missing” indicates constraints that the hierarchy category covered, which *not* covered by the induced rule, and “Extra” displays the number of additional constraints covered by the induced rule that were not included in the hierarchy category. In most cases, the induced rules represented the same constraints as those suggested by the hierarchy. However, there was one case where the outcome was not the same: for student A the set of constraints represented by “Function” contained twelve extra constraints and was missing six compared to the hand-coded hierarchy.

Student	Induced rule	Hierarchy category	Correctly classified	Missing	Extra
A	ORDERBY	Sorting	10	-	-
	Function	Aggregate Functions	38	6	12
	GROUPBY	Grouping	2	-	-
	Grouping	Grouping	0	-	-
	NULL + condition	Null / attributes	3	-	-
B	Place	Negation / correct attributes	0	-	-
	Another + table	All tables used / none missing / nested selects	0	-	-
C	Arithmetic	Expressions / arithmetic	5	-	-
	DISTINCT	Expressions / DISTINCT	0	-	-
	Grouping	Grouping / exists / having	0	-	-
	Another + table	All tables used / none missing / nested selects	0	-	-
	GROUPBY + Check	Grouping / attributes	0	-	-
	NOT+right	Negation / attributes	0	-	-
	Need + resulting	Sorting / existence	0	-	-

Table 3. Results for three students

The extra constraints arose because there were twelve more constraints concerning aggregate functions that were “hidden” in another part of the hand-coded hierarchy (“comparisons with constants”), and so were not included by it. This highlights the problem of having a single view of the constraints. The induced rule set is therefore superior to the hierarchy in this respect. However, the missing constraints are a genuine problem. Because categories are induced from free-format text, there is no guarantee that a consistent terminology will have been adopted. In this case the word “function” was used in most, *but not all*, of the constraints concerning aggregate functions. Hence, some constraints were missed.

Overall, the method performs quite well. The induced rules are very similar to those obtained by the hand-coded hierarchy and should be useful for problem selection. The lack of consistent terminology in the feedback messages poses a genuine threat to this method. However, its effect seems to be fairly small: some relevant constraints have been missed, but no constraints were incorrectly included. In any case a perfect result is not essential, since the effect of missing or adding extra constraints will at worst be a degradation of the gains in performance of problem selection.

The hand-coded hierarchy has clear benefits in other areas. Because the hierarchy was carefully chosen to be a meaningful abstraction of the constraints, it could be presented to the student to illustrate the structure of the domain, and similarly the student model could be presented to help both student and teacher understand where the problem areas lie. However, the induced rules are based purely on regularities in the textual feedback messages so the results are not always understandable in isolation: “Grouping” and “NULL” are understandable; “Place” is not.

Finally, both the hand-coded hierarchy and the induced rules might be used to select high-level feedback. In the case of the hierarchy, each node could have an appropriate message attached to it, which is displayed when the node describes the student’s behaviour. For the induced rules, a pool of extra messages could be provided at varying levels of generality. Then, as well as producing classification rules for “Not Learned”, a rule set could be produced for “Learned.” If a high-level message matches a rule for “Not Learned”, but does not match any for “Learned,” it is probably relevant to this student. Conversely, a message that matches *both* rule sets is probably too general.

6.2.6 Selecting the target constraints

We suggested that by inducing high-level concepts a student’s misconceptions could be determined from the text of the constraints they violated. This allows us to identify those concepts a student is finding difficult. These can then be used to guide the pedagogical process by aiding tasks such as next problem selection. We have shown that in the case of SQL-Tutor, the induced rules appear promising compared to the hierarchy we previously hand-coded, although further evaluation is required to verify

the method's performance. The set of constraints represented by the induced rules may now form the target set. Further, if a curriculum structure exists this might be used to reduce the target set to an individual concept, for example by only permitting constraints for a particular clause.

6.3 Building a new ideal solution

Each pattern match in the relevance condition of the target constraint corresponds to a fragment of the solution that must be present for this constraint to be relevant. We therefore begin by inserting these fragments into our (currently blank) ideal solution. Since the pattern matches may contain variables, these must be instantiated. In SQL-Tutor these variables may correspond to database table or attribute names, literals, relational operators etc. In some cases the value will be constrained by tests in the constraint, which resolve to a set of allowed values. For example, a variable representing a relational operator must contain a member of the set (>, <, <=, >=, =, <> or !=), so the algorithm may instantiate the variable to a random element of the set. A variable representing a database table will be similarly constrained to a member of the list of valid table names. In other cases (e.g. literals) there is no such set. However, such variables cannot be simply assigned a random value: in any given instance some values will be sensible, others will not. For example, if the subject of the database being queried is movies, the condition "`Title = 'Sparticus'`" would be sensible, but "`Title = 'sekfgdvfv'`" would not. To overcome this problem we introduce a small set of *instantiation* constraints, which further restricts the value of such literals. These constraints are used only during the production of new problems.

The instantiation constraints also ensure that semantic consistency is maintained, and may check that the new problem does not increase markedly in difficulty during the next phase (building a complete solution). As an example of the former, the movies database contains information about who stars in each film. A new problem might independently add two fragments for comparisons with a literal: one for the title of the movie and one for the name of the role being played by a particular star. A constraint is necessary to ensure we do not build obviously artificial conditions such

as in the following example, where the role of “Noddy” does not exist in the film “Star Wars”, so this problem would seem nonsensical to a movie-going student:

```
WHERE title = ’Star Wars’ and role = ’Noddy’
```

An example of where problem difficulty might escalate is in the (random) assigning of database attribute names to variables: each attribute could potentially come from a different database table. This would require the joining of many tables, which is one of the most difficult aspects of SQL. Therefore in the absence of a JOIN in the target constraint set, all attributes should come from a single table. Instantiation constraints achieve this. The following is an example of an instantiation constraint and its accompanying macro:

```
(I8
"Ensures that literal string comparisons in WHERE are with valid strings"

(match SS WHERE (?* (^attribute-in-from (?name ?attr ?table))
                    (^rel-p ?op) (^sql-stringp ?str) ?*))

(test SS (^valid-string (?attr ?table ?str)))

"WHERE")

(^valid-string (??attribute ??table ??string) =
    (test ?? ((
       ("lname" "director" "'Kubrick'")
       ("lname" "director" "'Spielberg'")
       ("fname" "director" "'Stanley'")
       ("fname" "director" "'Steven'")
       ("title" "movie"    "'Star Wars'")
       ("title" "movie"    "'Blazing Saddles'")
       ...
       (??attribute ??table ??string))
    )
)
```

Other instantiation constraints match multiple strings. For example, one constraint ensures that if `director.fname` and `director.lname` are both present, they are a matched pair such as `'Stanley'` and `'Kubrick'`. Another ensures that stars' names and their roles are consistent.

At this stage our new potential ideal solution consists of a set of disjoint fragments, which may or may not be valid SQL. They are now passed through the problem solver, which corrects any errors leaving a valid SQL solution. The algorithm used is identical to that designed for problem solving that was described in section 4. However, in this case only the syntactic constraints are used.

6.4 Controlling problem difficulty

The new problem must also be of appropriate difficulty. Brusilovsky (Brusilovsky 1992) suggests that tasks may be selected according to the combination of two independent measures: structural complexity and conceptual complexity. *Structural* complexity is a measure of how difficult a problem is *per se*. Brusilovsky defines it as the number of steps to solve a problem. *Conceptual* complexity is a measure of how much this problem requires the use of concepts the student has not yet mastered. He defines this as the number of "not quite learned" knowledge elements from the domain model.

To determine the next best problem to select, ITEM/IP (Brusilovsky 1992) adds two variables to the student model: the current optimal structural complexity, and the current optimal conceptual complexity. Both are dynamic: if the student solves a problem, the student model complexities are set to the maximum of the current values and those for the newly solved problem; if the problem is not solved, they are reduced. To select the next problem ITEM/IP first compiles a list of problems that are eligible (i.e. all of the skills are learned sufficiently to be ready to practice, and at least one is not fully learned yet). It then selects the best one by minimising the difference between problem complexity and the student's current optimal complexity. This difference is defined as:

$$Diff = W_1\left(SC_p - SC_s\right)^2 + W_2\left(CC_p - CC_s\right) + W_3\left(Err\right) \qquad (4)$$

where *SCp* and *SCs* are the structural complexities for the problem and student respectively, *CCp* and *CCs* are the corresponding conceptual complexities, and *Err* is the number of erroneous tasks required, i.e. those that are not relevant to the current curriculum topic. Although Brusilovsky does not specify the weights, it is clear that the structural complexity is the dominating term. He reports that the described method has been used in systems for both first year university students and 14 year-old school pupils. In both cases the students found the task sequencing strategy "seemed intelligent, and they usually agreed with the system's choice" (Brusilovsky 1992). We set the values of the three weights empirically, by trying different values and

observing how well the system stayed on a concept for which errors had been made, how quickly it moved on to a new concept once it was mastered, and how many new concepts it introduced at a time. For SQL-Tutor we used W1=1, W2=5 and W3=10, making a single failed constraint the equivalent of five known ones, and favouring staying on the current failed constraint over moving to a new (previously not encountered) constraint.

In SQL-Tutor there is no concept of number of problem-solving steps required. Instead the domain model is built around the underlying domain concepts that are involved in a problem's solution, which translate into constructs present in the completed solution. We therefore use this as a basis for computing complexity. Factors that might affect structural complexity are therefore:

1. The total number of constructs involved;
2. The number of new constructs;
3. The number of not learned constructs (i.e. those the student is likely to fail, based on previous experience);
4. The complexity of each construct in (1, 2, and 3).

Note that a much simpler scheme for calculating structural complexity might be to count the number of terms in the solution. However, this ignores the fact that a single complex term (such as a JOIN or nested select) is likely to add much more difficulty than, say, three WHERE conditions involving straightforward comparisons with literals.

ITEM/IP uses simple counts of the number of tasks. In the case of SQL-Tutor the complexity of each construct must also be taken into account, since not all constructs are the same. There are two ways we could assign difficulty to the constructs associated with constraints: by manually assessing difficulty, or by automatically assigning a value for each constraint. Since we are trying to minimise the work required to build tutors, we chose the latter, although we concede that this is a compromise, since some constructs may be considerably more difficult to build than their surface complexity implies. We calculate the complexity of each construct from its size: the larger the construct, the more difficult it is likely to be. We use the sum of terms squared as the complexity measure, the same as Brusilovsky, i.e.

$$\text{Difficulty} = n^2 \quad (5)$$

where n is the number of terms introduced, which is equal to the number of non-wildcard elements in the MATCH fragment(s) added as a result of this constraint. The difficulty is computed continually as the solution is built up: when new fragments are added, the complexities for the added fragments are added; when fragments are deleted, they are subtracted. A TEST modification is equivalent to a MATCH with one term.

Conceptual complexity measures the degree to which the student is likely to struggle with the new concepts introduced in this problem. Again Brusilovsky used the number of tasks required that have not been learned yet. Instead, we use the total complexity of new constraints introduced that are from the target constraint set. For the measure of "erroneous" concepts, we total the complexity of all relevant constraints that have never been encountered before. Conceptual complexity is measured in the same way as structural complexity but only those constraints that have previously been violated are used in the summation process, which continues until the solution is empty or the candidate constraints have been exhausted.

Whereas ITEM/IP records the student's ideal conceptual and structural complexities in the student model, we record a single value of optimum difficulty and compare the difficulty of each problem with respect to the student model to this value. For a new student model the target difficulty is set to an initial value, which depends on the competency level the user selects when they log in. To determine what value to set each competency level, we computed the difficulty of the existing authored problem set. This set ranged in difficulty from 6 to 1084, with a mean of 240 and standard deviation of 264. We adjusted these figures slightly so that the existing problem set was partitioned sensibly, to give default difficult difficulties of: Novice = 0, Average = 250 (approximate mean), and advanced = 500 (approximately the mean + 1 SD).

The student's target difficulty value is updated each time they complete a problem-solving activity. If the student successfully solved the problem without help, the variable is incremented by a constant amount, K_1. Similarly, if the student fails,

the system decreases their levels by K_2. The values of K_1 and K_2 were ad hoc, being K_1 = 50, and K_2 = 10. This means that the target complexity rises quickly if the student answers a problem correctly, but falls slowly if they fail consecutive ones. This ensures that the difficulty of problems being set does not trend too quickly to zero because a student is struggling with some concept.

Problems are constructed to match the target complexities as follows:

- A target constraint is selected. To ensure that difficulty is appropriate, we select the simplest target constraint that meets or exceeds the student's ideal conceptual complexity. The complexity for the solution so far is then updated according to the number of unmatched terms in the added fragment. If the most complex target constraint fails to meet the required complexity, further constraints are added in the same manner, until the desired conceptual complexity is reached. If, at the end of this procedure, the problem is still not sufficiently complex, we select the next target constraint set, and continue until either the desired complexity is reached, or the set of target constraint sets has been exhausted;
- Further constraint fragments are added from the set of learned constraints. Each time a new construct is included, its complexity is added to the total, i.e. the square of the number of terms in the total fragment;
- During the final building of a correct SQL statement, the same scheme is applied, i.e.
    - Every time a fragment is added, the complexity for the extra terms is added to the total;
    - Every time a fragment is deleted, the complexity for the removed fragment is deducted.

Note that the third step (building a complete solution) may further increase the structural and conceptual complexities of the solution. This is minimised by the instantiation constraints, which attempt to keep the overall structure of the solution as simple as possible while satisfying the target constraint(s).

6.5 Converting to natural language

The final step is to produce a natural language problem statement for which the newly generated SQL statement is a correct answer. Again a small set of constraints is used, which maps constructs in the SQL statement to a Natural Language representation of the problem to be solved. As before, multiple ways of representing any part of the problem are catered for, allowing variation in problem phrasing. The problem statement is structured in a similar fashion to an SQL query, in that each will contain a phrase that describes the attributes to be selected, another for which entity(ies) these attributes belong to etc. These phrases are concatenated to give the complete problem statement. For example, the following three constraints help generate the first phrase, i.e. which attribute(s) to retrieve:

```
(NLP1 "selects a random intro for the ATTRIBUTES phrase"
(match IS SELECT (?what ?*))
(match PROBLEM ATTR-HDR
   (("List all") ("Produce a list of") ("what is the")) ?heading)))
"")

(NLP2 "translates all attributes in the SELECT clause into a
suitable synonym"
(match IS SELECT (?* (^attr-synonym (?n ?s)) ?*))
(match PROBLEM ATTRIBUTES (?* ?s ?*))
"")

(NLP3 "Makes sure there is a comma between attributes"
(and (match PROBLEM ATTRIBUTES (?* ?s1 ?s2 ?s3 ?*))
     (not-p (test PROBLEM (“,” ?s1))) (not-p (test PROBLEM ("," 
?s2)))
)
(and (match PROBLEM ATTRIBUTES (?* ?s1 “,” ?s2 ?s3 ?*))
       (not-p (match PROBLEM ATTR (?* ?s1 ?s2 ?s3 ?*)))
)
"")
```

The number of NLP constraints will depend on the complexity of the domain and the flexibility in language required. For example, the following SQL problem could be stated in several ways:

```
SELECT lname, fname
FROM star
WHERE born >= 1920 and born <1930
```

This could be mechanically translated into “List the last name and first name of all stars where born is at least 1920 and less than 1930”. However, a more natural

statement for this problem, which would “give away” less of the solution, is “What are the names of all stars born in the twenties?” The latter would require a considerably more sophisticated constraint set to cope with for example, the fact that the attribute “born” is now being used as a verb. We estimate that SQL-Tutor would require a minimum of around 25 constraints to mechanically translate queries into SQL, and at least 100 to demonstrate suitable flexibility to be able to recreate the current human-authored problem statements. Both would also require taxonomies to translate attribute names, table names, comparison operators etc into natural English.

6.6 Problem generation example

During the study described in (Martin 1999), we examined the state of several students at the conclusion of a two-hour session with SQL-Tutor. Student A was found to be still failing constraints concerning sorting, aggregate functions, grouping and null attribute tests. Suppose we wish to generate a new problem to test sorting. We select a constraint at random from the induced target set, for example:

```
(378
"Check whether you should have ascending or descending order in the
ORDER BY clause."
(and (match IS ORDER_BY (?* ?n "DESC" ?*))
     (match SS ORDER_BY (?* ?n ?*))
)
(match SS ORDER_BY (?* ?n "DESC" ?*))
"ORDER BY")
```

From this constraint, we obtain the fragment `ORDER_BY (?n DESC)`. Student A is an average student, so we need to increase the difficulty of the problem to her level. We randomly select one or more constraints that she has already learned, for example:

```
(175
"Check you are comparing the string constant to the right attribute
in WHERE."
(and
  (match IS WHERE (?* (^attr-name (?n1 ?a1 ?t1)) (^rel-p ?op1)
(^stringp ?c)
                   ?*))
  (match SS WHERE (?* (^attr-name (?n2 ?a2 ?t2)) (^rel-p ?op2) ?c
?*))
)
(test SS (^same-attributes (?a2 ?t2 ?a1 ?t1)))
"WHERE")
```

Because this is a semantic constraint, we use the first ideal solution match, which adds a comparison between an attribute and a string. At this stage the attribute variable will be randomly instantiated to a valid attribute and the relational operator will similarly be instantiated to one of the relational operators. The string cannot be instantiated yet. This partial solution is then passed to the instantiation constraints. Since the FROM clause is empty, it is instantiated to a random valid table name. Now the attribute in the WHERE clause must be a valid attribute from that table, so it is modified if necessary. Finally the attribute/string pair must be a valid pairing as defined in the instantiation attributes. The solution thus far is now (for example):

```
FROM customer
WHERE lname = ’Parker’
ORDER_BY ?n DESC
```

This partial solution is now passed to the problem solver, which uses the syntactic constraints to build a valid SQL solution:

```
SELECT number
FROM customer
WHERE lname = 'Parker'
ORDER_BY number DESC
```

Note that on completion of this stage it is possible that the generated solution will violate the instantiation constraints (for example, by introducing an additional table name) and/or alter the difficulty of the problem unacceptably by adding or removing fragments. It is therefore necessary to re-test both of these aspects. In the case above, no further modification is necessary.

Finally, the generated ideal solution is converted into a natural language problem statement using the constraints designed for this purpose, for example:

List all numbers of customers whose last name is Parker. Order the results by descending customer number.

6.7 The problem generation algorithm

In the preceding sections, problems were generated online each time the student concludes an exercise. The high-level algorithm is:

1. Update the student model complexity variables according to the student's performance on the latest exercise;
2. Use the ML algorithm to induce a new set of target constraints;
3. Choose a target constraint, and insert its corresponding fragment(s) into the (currently blank) new ideal solution;
4. Choose additional constraints, and insert their fragments, until the desired level of difficulty has been achieved;
5. Test the solution against the instantiation constraints;
6. Build a complete solution, using the problem solver;
7. Convert to a natural language problem statement;
8. Present to the student.

This approach has one major disadvantage: it requires that both the solution and problem generation algorithms be fail-safe. As seen in Chapter 5, this is not an easy task. For solution generation, testing the constraint set to ensure that all incorrect solutions will be corrected is difficult if not impossible. In problem generation, the problem is worse. First, we are now considering building SQL without a clear semantic requirement, so it is even more likely that the algorithm will generate mistakes that a student is highly unlikely to do. Second, to avoid nonsensical questions the initialisation constraints also need to be infallible. Finally, generating plausible natural language queries that do not make the solution obvious is difficult. In solution generation, the most common problem is that the algorithm fails to terminate. This can be trapped and the fallback position adopted where the ideal solution is simply used. There is no such parallel in problem generation: how can the system trap a nonsensical problem?

An alternative is to perform problem generation offline. In this scenario the problem generation algorithm is used to (try to) create *n* problems per constraint. On completion, a human teacher assesses the generated problems and decides which ones to keep, and which to discard. She may also alter some problems to improve their semantics. The created problem set is then used. Problems are now selected (rather

than generated) by comparing all problems to the student model to determine their conceptual difficulty, as described previously. Whichever is the closest fit is selected for presentation. During generation the difficulty of each problem may be controlled as described previously to ensure sufficient spread of difficulties. Alternatively, the problems can simply be built with no regard to difficulty, relying instead on the variation in constraint difficulties to ensure an even spread. In our experiments we chose the latter.

The algorithm loops through each of the constraints trying to build *n* new problems. Note that some constraints test for the absence of erroneous constructs, so can never be successfully turned into problems. Rather than waste time determining that this is the case, these are explicitly excluded from processing. We add a fragment for each constraint based on either the relevance condition, or a *prototype*. Some constraints test only a very small part of a larger construct. To build a full SQL query for such a constraint relies on the other constraints for this construct being successfully applied. However, because the original fragment is such a small part of the construct, it may fail to be correctly identified by the constraint set, which may turn it into a different construct, or possibly delete it. Consider the following constraint:

```
(439_A
   "Make sure you are using the right kind of JOIN."
   (and
        (match IS FROM (?* (("LEFT" "RIGHT") ?jt1) ?*))
        (match SS FROM (?* (("FULL" "LEFT" "RIGHT") ?jt2) ?*))
   )
   (test SS (("LEFT" "RIGHT") ?jt2))
   "FROM"
)
```

This constraint tests that the correct type of JOIN is being used. However, it only specifies the type of join *and nothing else* in the tests of the ideal solution in the relevance condition. It therefore adds only the fragment (for example) "`LEFT`" to the FROM clause. This by itself is not valid and, since the instantiation constraints favour a single table over multiple ones, the lone "`LEFT`" is assumed to be a mistake and deleted by another constraint. To overcome this, we add the following prototype to the constraint, which builds a complete JOIN condition:

```
; PROTOTYPE
(and
   (match IS FROM (?* ?t1 (("LEFT" "RIGHT") ?jt)
      "JOIN" ?t2 "ON" ?* ?n1 "=" ?n2 ?*))
   (test IS (^join-fields (?n1 ?t1 ?n2 ?t2)))
      )

)
```

Note that prototypes are not strictly necessary: we could also make the ideal conditions of the relevance conditions more stringent and achieve the same end. For example, in the constraint given we might ensure that the test of the ideal solution contains a full JOIN construct, since it would be incorrect without it anyway. However, for some constraints this is not such an obvious step, and it confuses the tasks of writing the constraints and facilitating problem generation. We began using this latter method, but found it much simpler to use prototypes. Of the 819 constraints in SQL-Tutor, 312 have prototypes. However, many of the prototypes are identical. For example, there is a prototype for building a nested query with appropriate attributes and tables that is used 54 times.

For those constraints without a prototype, the relevance condition is used. For semantic constraints we are interested only in the ideal solution, so we rename all IS (ideal solution) matches and tests to SS (student solution) for passing to the problem solver, and delete all the original student solution tests. Syntactic constraints are tested verbatim. A summary of the algorithm is given in Figure 9.

```
To generate a set of problems
Open log file
Open problem file

Instantiate a list of constraints that are never relevant if the
solution is correct

For all constraints (except those in the not-relevant list)
    Try to generate a problem from the constraint
    If successful (relevant to the constraint)
        Write out to the problem file
    ELSE
        Report that it failed to resolve to a problem

Close the files

To generate a problem:
Add a fragment for the constraint into the (currently blank) solution

Try to correct the solution using solution generation algorithm, but
alternating testing the instantiation constraints and syntactic
constraints, instead of the semantic and syntactic constraints.

To add a fragment:
try 20 times max:
If the constraint has a prototype, set CONDITION to that
ELSE
set CONDITION to the relevance condition

If the constraint is semantic, rename all IS tests in CONDITION to be
SS tests, and delete the SS tests

Set bindings to be a default root (i.e. the default binding for a
successful evaluation, where no variables were encountered)

Evaluate CONDITION as though it is a satisfaction condition

If there are corrections to be performed
      action them
ELSE
    Return the solution unchanged
```

Figure 9. Problem generation algorithm

6.8 Evaluation

The motivation for Problem Generation was to reduce the effort involved in building tutoring systems by automating one of the more time-consuming functions: writing the problem set. Three criteria must be met to achieve this goal: the algorithm must

work (i.e. it must generate new problems); it must require (substantially) less human involvement than traditional problem authoring; and the problems produced must be shown to facilitate learning to at least the same degree as human-authored problems. These criteria form the basis of three hypotheses that must be supported by empirical evidence before we can be satisfied that the method is worthwhile. Additionally, if the method works, it should be possible to generate large problem sets, which will have the benefit of greater choice when trying to fit a problem to the user's current student model. Also, the new representation allows us to measure the structural difficulty of each problem as previously described. This allows us to measure the *conceptual* difficulty of each problem in relation to the student model, and thus to choose the best problem for this unique student model, rather than simply basing our choice on the student's aptitude level. We might therefore expect that, given a suitable problem selection strategy, a system using the generated problem set would lead to faster learning than the current human-authored set and high-level problem-selection strategy. This gave us four hypotheses to test:

Hypothesis 6.1: That the algorithm successfully generates new problems;

Hypothesis 6.2: That generating new problems is easier than authoring them manually;

Hypothesis 6.3: That using generated, rather than human-authored, problems does not significantly degrade performance of the ITS;

Hypothesis 6.4: That by using a problem selection routine that takes advantage of the new representation's ability to calculate conceptual difficulty, the new problems (plus the new selection routine) may lead to an *increase* in learning performance.

We tested the first two hypotheses in the laboratory, while the last two were evaluated using a university class.

6.8.1 Testing of hypotheses 6.1 and 6.2

The SQL-Tutor ITS was used as the basis for all testing of Problem Generation. The knowledge base created for problem solving in Chapter 5 formed the basis for testing.

Additionally, 66 instantiation constraints were added for controlling the semantics. The problems were generated in batch using the algorithm described in Section 6.7. At this stage, the natural language converter was not implemented, so human intervention was necessary to produce the text message for each problem. To keep the number of problems manageable, the batch run was limited to one problem per constraint. We generated a total of 819 problems.

We then checked each problem for sensibility and, if accepted, authored the problem text message. Of the 819 problems, approximately half emanated from semantic constraints, with the other half being syntactic. Because of time constraints, we elected to use only those from the semantic constraints. Of these, 200 were chosen that had sensible semantics and were not duplicates, or around 50%. In practise, nearly all of the rejections were because the generated SQL was nonsensical. Some examples of reasons for rejection are:

- The problem contained illegal combinations of literals, caused by deficiencies in the instantiation constraints;
- The structure of the problem was unrealistic (e.g. double negatives);
- The problem was testing some unusual construct that was unlikely to teach anything useful;
- The ideal solution was identical to another;
- The problem when converted to text would have been identical to another (i.e. the ideal solution was different but the semantics were the same).

The process of vetting the problems and producing text input for all of the problems took a total of approximately three hours, compared to many days to author the 82 problems manually in SQL-Tutor, so it took much less time to produce 200 problems in this way than to manually author 82. Note that this rate of authoring is atypical: this author produced these problems, so had the benefit of deep immersion in both the domain and the generation process, and had knowledge of the types of problems that would be generated, and likely difficulties with them. Further, simple

deficiencies (such as problems involving double negatives) appeared in large blocks because the constraints tend to be grouped by function. For example, there is a large block of constraints that deals with NOT. Thus, double negatives tended to appear close together. Finally, there was a reasonable level of structural similarity between blocks of problems, for example many problems dealing with nested queries were grouped together. This may not always be typical of a domain knowledge base. In spite of these caveats, we still believe problem generation will save a significant amount of time when authoring other domains: the author of the original problem set (Mitrovic) was similarly immersed in both the subject: she is a teacher of database material, and she authored the original SQL-Tutor system.

The generated problems were used successfully on a university class (see next section). Hypotheses 6.1 and 6.2 are therefore supported in the SQL domain: the algorithm worked, and it took substantially less time to author problems using it than creating them manually.

6.8.2 Classroom evaluation of hypotheses 6.3 and 6.4

To test hypotheses 6.3 and 6.4 it is necessary to demonstrate that a system using generated problems performs as well as or better than a system using human-authored exercises. We modified SQL-Tutor for this purpose and evaluated it for a six-week period. The subjects were stage two university students studying a databases paper. At the end of the study the students were required to sit a lab test about SQL as part of their assessment, so they were motivated to use the system if they considered it might improve their performance.

We partitioned the students into three groups. The first used the current version of SQL-Tutor, i.e. with human-authored problems. The second group used a version with problems generated and selected using the algorithms described. The third group used a variant containing other research (student model visualisation) that was not relevant to this thesis. Each student was randomly assigned a “mode” that determined which version of the system they would use. Before using the system each student sat a pre-test to determine their existing knowledge and skill in writing SQL queries. They were then free to use the system as little or as often as they liked over a six week

period. At the conclusion of the evaluation they sat a post-test. Appendix A contains the pre- and post-test scripts.

When the study commenced approximately 60 students had signed up and performed the pre-test, giving sample sizes of 20 per group. During the evaluation this swelled to around 30 students per group as new students requested access to the system. At the conclusion of the study some students who signed up had not used the system to any significant degree. The final groups used for analysis numbered around 20 students each. The length of time each student used the system varied greatly from not using it at all to around twenty hours, with an average of two-and-a-half hours. Consequently the number of problems solved also varied widely, from zero to 98, with an average of 25. Thus, an average student might expect to learn the domain in under three hours, with struggling students taking considerably longer. Three of the students in the evaluation study solved more than 82 problems, the total number available to the control set. These figures are discussed in more detail in Section 6.8.4. At the end of the six-week period we closed the student logs and analysed the results. We recorded the following information in the logs:

- A timestamp for each action;
- The problem number;
- The student's attempt;
- A list of the violated and satisfied constraints;
- On selection of a new problem, the student difficulty and the difficulty of the chosen problem;
- On aborting a problem, the reason for failing to finish (if entered by the user).

From this information we deduced summary information such as the status of each constraint over time, the time spent on each problem attempt and the number of attempts per problem. We used these results to analyse how each version of the system supported learning.

There are several ways we can measure students' performance while using each system. First, we can measure the means of the pre-test and post-test to determine whether or not the systems had differing effects on test performance. Note however, that with such an open evaluation as this it is dangerous to assume that differences are

due to the system, since use of the system may represent only a portion of the effort the student spent learning SQL. Nevertheless, it is important to analyse the *pre-test* scores to determine whether the study groups are comparable samples of the population.

Second, we can plot the reduction in error rates as the student practices on each constraint. Each student's performance measured this way should lead to a so-called "Power law" (Newell and Rosenbloom 1981), which is typical when the underlying objects being measured (in this case constraints) represent concepts being learned. The steepness of this curve at the start is a rough indication of the speed with which each student is learning new constraints. Since each constraint represents a specific concept in the domain, this is an indication of how quickly the student is learning the subject. We can then compare this learning rate between the two groups.

Third, we can measure how many new constraints the student is introduced to, and masters, each time they solve a new problem. The higher the number, the more likely that the student is learning faster. If the problems are well suited to the student's current abilities, the system should be able to introduce more new material without overloading the student, because the new material is relevant to what the student already knows and is of an appropriate level of difficulty.

Finally, we can look at how difficult the students found the problems. This is necessary to ensure that the newly generated problems did not negatively impact problem difficulty (either by being too easy or too hard). There are several ways we can do this. First, we can measure how many attempts the student took on average to solve a problem and compare the means for the control and test groups. Second, we can measure the time taken to solve each problem. Note however, that this is an extremely crude measurement, since it does not take into account any "idle" time Note, however, that it does not include idle time at the end of the session, since this is not followed by a solution being submitted, so is not counted. Finally, students may abort the current problem, citing one of three reasons: it was too easy, it was too hard or they wanted to try a problem of a different type. If the proportion of problems aborted rises or the ratio of "too hard" to "too-easy" problems is very different to 1:1, we might conclude that problem difficulty has been adversely affected.

Group	Aborted (%)	Too hard (%)	Too easy (%)	Diff Type (%)	Responded (%)
Control	26	24	42	34	84
Probgen	26	22	42	35	62

Table 5. Aborted problems

In this study, we measured all of the above. We used the software package SPSS to compare means and estimate power and effect size, and Microsoft Excel to fit power curves. We now present the results.

6.8.3 Pre- and post-test performance

Each student was given a unique (anonymous) username, which was used to correlate the pre- and post-tests. Unfortunately, not all students who used the system provided their username on the post-test, so the sample sizes were reduced to 12 and 14, respectively. We measured the means for the pre-test, post-test and the gain (i.e. the difference between the pre- and post-test results. Table 4 lists the results (standard deviations are in parentheses). We measured significance using an independent samples T-test (two-tail). These indicate there was no significant difference in performance between the groups for either the pre-test or the post-test, nor in the gain observed for each student. As already mentioned, such an open evaluation is unlikely to show significant results because we do not know what other effort the students expended to learn SQL. Also, since the pre- and post-tests were different, it is possible that they are not comparable, e.g. the post-test may be much harder or may favour a particular type of problem that one system set more problems on. However, because the pre-test means were not significantly different, we can assume that the samples are comparable, which validates the rest of this study.

6.8.4 Problem difficulty

We measured problem difficulty both subjectively and objectively. We obtained

Group	Sample	Pretest	Posttest	Gain
Control	12	4.41 (1.24)	6.42 (1.38)	2.00 (1.34)
Probgen	14	5.21 (1.31)	6.79 (1.37)	1.57 (2.31)
Significant?		No ($p = 0.12$)	No ($p = 0.50$)	No ($p = 0.56$)

Table 4. Test score results

subjective results by logging when students aborted a problem and recording their reason. If the problems were (overall) of a suitable difficulty, we would expect the ratio of claims of “too hard” to “too easy” to be approximately 1:1. Any significant move away from this ratio would indicate we have adversely affected problem difficulty. Further, the percentage of problems aborted should not rise significantly. Table 5 lists the results. “Aborted” indicates the proportion of all problems attempted that were aborted. “Too hard”, “Too easy” and “Diff type” give the proportion of *aborted* problems for which the reason given was the problem was too hard, too easy, or the student wanted a problem of a different type, respectively. “Responded” indicates the proportion of aborted problems for which the student gave a reason. This last measure was different for the two groups, with a lower response rate for the Problem Generation group. We do not know why this difference occurred, since the interface was the same, and the other factors were not changed.

These results suggest that for both groups the problems set are more often too easy than too hard. The percentage of problems aborted in each group was exactly the same, at around 26% of all problems. The ratio of too easy to too hard for the two groups is nearly identical, as is the proportion of problems aborted because the student wanted a problem of a different type. It therefore appears that the generated problems had no effect on difficulty as perceived by the students.

Next we measured the number of attempts taken to solve each problem. This gives an objective indication of how hard students found the problems. Table 6 lists the results. “Solved/student” indicates the average number of problems completed correctly. “Total time” is the average time spent at the system. This figure records the time that the user was actively using the system, from when they first logged in to when they last submitted an attempt. Thus it excludes idle time where the user has forgotten to log out. “Attempts per problem” is the ratio of submitted attempts to solved problems for *all* attempts, including those for problems the student abandoned. The rationale is that attempts at unsolved problems still constitute a learning effort, so should be counted as effort towards the problems that were actually solved. “Time/Problem” similarly records the total time spent on the system divided by the number of problems solved, using the same rationale.

Group	Solved/ Student	Total Time (hrs)	Attempts/ problem	Time/Problem (mins)
Control	23	2:37	3.96 (1.89)	6:14 (3:58)
Probgen	26	2:31	3.45 (1.23)	5:50 (3:20)
Significant?			No (p = 0.31)	No (p = 0.71)

Table 6. Attempts per problem

There was no significant difference in the objective measurement of problem difficulty: students took approximately the same amount of time and number of attempts in both groups. The number of problems solved and average total time spent on the system was also almost the same for the two groups, suggesting that students did not favour either system.

6.8.5 Learning speed

We observed the learning rate for each group by plotting the proportion of constraints that are violated for the *n*th problem for which this constraint is relevant. This value is obtained by determining for each constraint whether it is correctly or incorrectly applied to the nth problem for which it is relevant. A constraint is correctly applied if it is relevant at any time during solving of this problem, and is *always* satisfied for the duration of this problem. Constraints that are relevant but are violated one or more times during solving of this problem are labelled erroneous. The value plotted is the proportion of all constraints relevant to the nth problem that are erroneous. The value for n=1, therefore, is the ratio of constraints that were erroneously applied to the first problem to the number of constraints that were relevant to one or more problems: the value for n=10 is the ratio of erroneous to total constraints relevant for 10 or more problems, and so on.

If the unit being measured (constraints in this case) is a valid abstraction of what is being learned, we expect to see a "power curve". In (Mitrovic and Ohlsson 1999) this has already been shown to be the case. We therefore fitted a power curve to the each plot, giving an equation for the curve where the initial learning rate is determined by the slope of the power curve at X=1. Note that as the curve progresses, learning becomes swamped by random erroneous behaviour such as slips. In other words, the plot stops trending along the power curve and levels out at the level of random mistakes. This is exacerbated by the fact that the number of constraints being

considered reduces as n increases, because many constraints are only relevant to a small number of problems. We therefore use only the initial part of the curve to calculate the learning rate. Figure 10 and Figure 11 show two such plots, where each line is the learning curve for the entire group on average, i.e. the proportion of constraints that are relevant to the first problem that are incorrectly applied *by any student in the group*. The first uses a cut-off of n=40, to illustrate how the curve tapers off. For the second, the cut-off was chosen at n=5, which is the point at which the power curve fit for both groups is maximal.

Both groups exhibit a very good fit to a power curve. The differing slope of the curves suggests a difference in learning rates between the problem generation and control groups. To determine whether this difference is significant, we plotted curves for each student and used this to measure their individual initial learning rates. This increases the effect of errors even further, so we determined empirically the best cut-off point for each group, which was found to be n=4. Figure 12 shows some of the plots obtained, ranging from very good power curve fit to poor. Low power curve fit almost always coincided with low learning performance.

We calculated the learning rate at n=1 for each student, and calculated the mean and significance. Table 7 summarises the results. These results suggest the learning rate for the experimental group was around double that for the control group. The effect is significant at α=0.05, p=0.01. A further test of the results is effect size and power. Using type 3 sum of squares testing (Chin 2001) we are striving for an effect size of 0.2, and a power (repeatability) of 0.8, i.e. an 80% likelihood of reproducing this result using the same experimental conditions. We obtained an effect size (omega squared) of 0.21, with a power of 0.794 at α=0.05, which is a very respectable result.

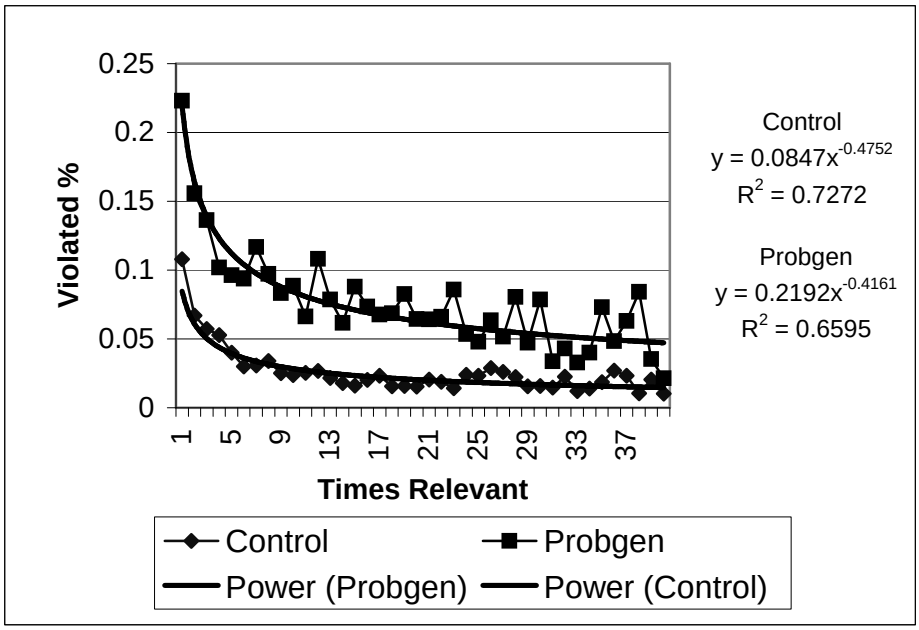

Figure 10. Learning curves, cut-off = 40 problems

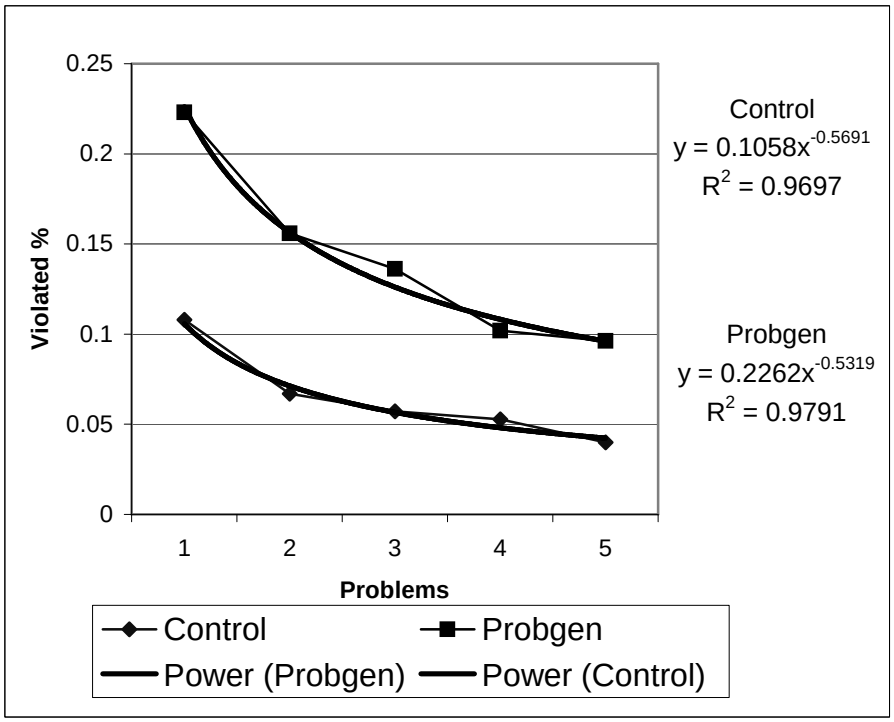

Figure 11. Learning curves, cut-off = 5 problems

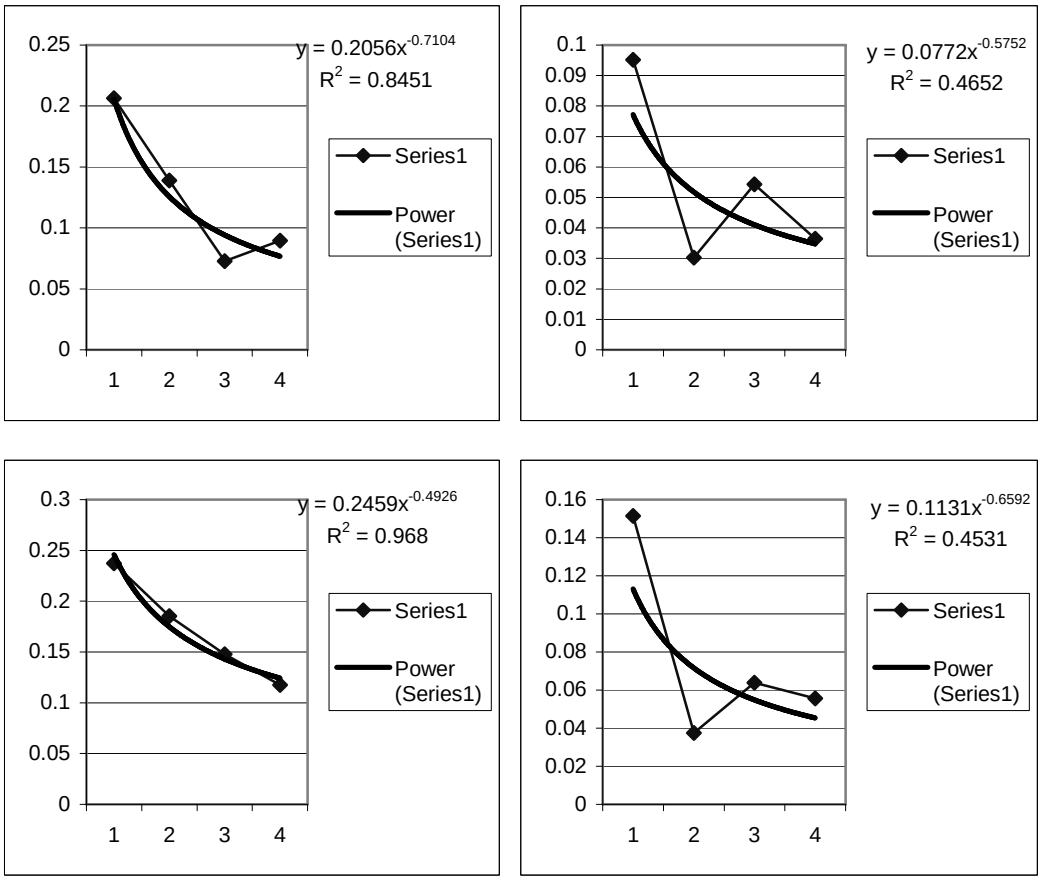

Figure 12. Examples of individual learning curves

A potential problem with comparing the control and experimental groups is that the constraint set is not the same, there being two significant differences. First, the control group uses a constraint set where around 80 of the constraints are trivially relevant. Of these, many are trivially true. For example:

```
'(p 364
   "You have used the backquote character (`) in
    SELECT. If you want to specify a constant, use
    a quote instead (')."
   t
   (null (search "`" (sel-txt ss) :test 'equalp))
   "SELECT")
```

This constraint is trivially satisfied, unless the student specifically uses a back quote.

Group	Slope	Fit (R^2)
Control	0.07 (0.04)	0.63 (0.29)
Probgen	0.16 (0.12)	0.68 (0.30)
T-test significant?	Yes (p = 0.01)	No (p = 0.61)

Table 7. Learning rates for individual students

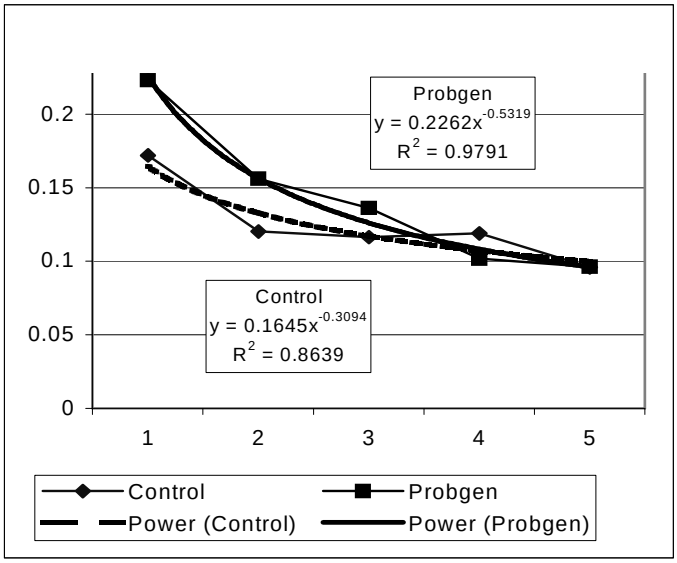

Figure 13. Error rates excluding constraints that are always true

In the experimental group, such constraints were rewritten such that by default they are not relevant. The effect is that there is a large body of constraints that are almost always satisfied in the control group irrespective of the student's behaviour, while in the experimental group they are absent. This may have an effect on the slope of the power curves. To verify that this is *not* the case, we removed all of these constraints and plotted the curve for the experimental group again. Figure 13 illustrates what happens: the control group curve shifts upward and becomes less smooth because the body of trivially true constraints normally has a smoothing effect. However, the slope of the curve is unchanged.

A second difference is that there are more constraints in the model for the experimental group, because it includes new ones added for solution generation. This would only be a problem if the new constraints were more likely to be violated than the rest of the set on average. In fact, most of the new constraints cover rare situations that only occur during problems or solution generation as a consequence of erroneous structures being built during correction, such as the existence of both the correct and incorrect versions of some construct. They are therefore unlikely to be relevant to a student solution, and so will fail to have any significance effect on the curves. We tentatively tested this assumption by running the answers submitted by the control group through the constraint evaluator for the experimental group, thus measuring the

Group	Slope	Fit (R^2)
Control	0.07 (0.07)	0.54 (0.30)
Problem Generation	0.16 (0.12)	0.68 (0.29)
T-test significant?	Yes (p = 0.04)	No (p = 0.27)

Table 8. Learning rates for individual students: new constraint set

students' progress in ability by exactly the same means as was used for the experimental group. Figure 14 shows the resulting learning curves for each group as a whole. The curve for the control group still has a lower initial slope than the experimental group, although the difference is less (0.12 for the experimental group, 0.078 for the control group). The power curve fit has also deteriorated. This is probably because the differing constraint sets means that the feedback received by the student no longer matches the constraints violated, hence there is some level of randomness creeping in. However, the fact that the experimental group still displays a higher initial learning rate indicates that the effect is not simply due to the differing constraint sets.

Finally, we recalculated the mean initial slope and fit from individual curves, again using the new evaluator. Note that the number of participating students for the control group shrank from 16 to 10, and the number of problems per student also shrank, because the system for the experimental group was implemented for one domain (MOVIES) only due to time constraints, hence problems for other domains (and all problems *after* one for another domain) were deleted from the logs. This accounts for the decrease in the average goodness of fit of the power curves (from 0.64 down to 0.54).

Once again, there is a statistically significant difference in the initial learning rate between the control and experimental groups at $\alpha = 0.05$ ($p = 0.04$), summarised in Table 8. In fact, this result is very similar to that achieved when the control group used the original constraint set. We can therefore assert with confidence that the difference is *not* because of the constraints.

A further indication of increased learning is the rate at which new constraints are introduced and successfully applied by the students. Figure 15 plots the number of constraints each student has demonstrated they have mastery of, versus the number of problems they successfully completed. In each case, the first plot shows the full curve,

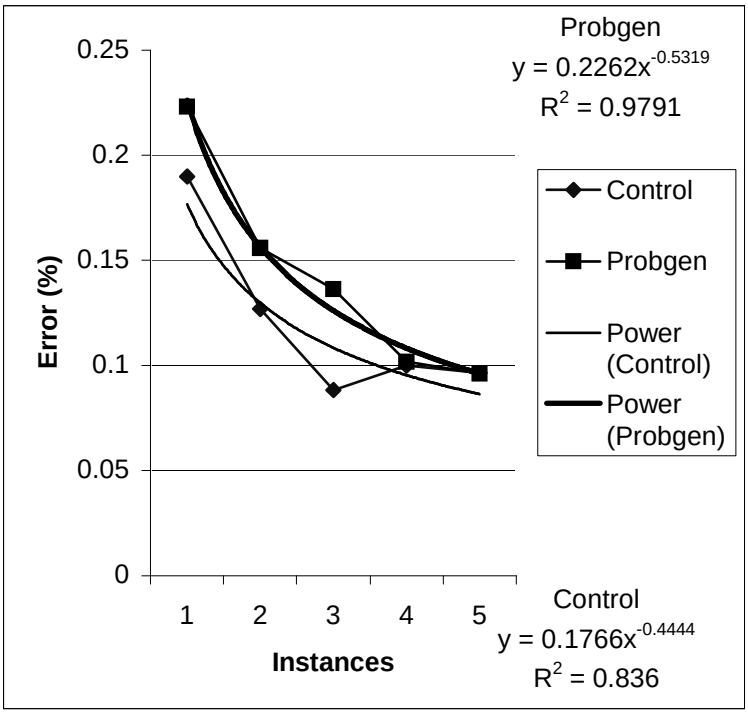

Figure 14. Learning Curves using the new evaluator for both groups

while the second is cut off at the point where the number of constraints introduced per problem suddenly tapers off. This effect occurs because the system has run out of problems of sufficient difficulty to give the student. The curves suggest that the generated problems successfully introduce more constraints per problem that the student is able to master.

To determine whether this effect is significant, we calculated the average number of constraints learned per problem per student for each group (after subtracting the X intercept from the above regressions), and calculated the significance. Table 9 summarises the results. They suggest that the generated problems introduced more constraints that were mastered per problem, although the results were not statistically

Group	**Constraints per Problem**
Control	2.51 (3.11)
Experimental	3.94 (1.30)
Significant?	No (p = 0.113)

Table 9. Constraints mastered per problem

significant at $\alpha = 0.05$ ($p = 0.113$).

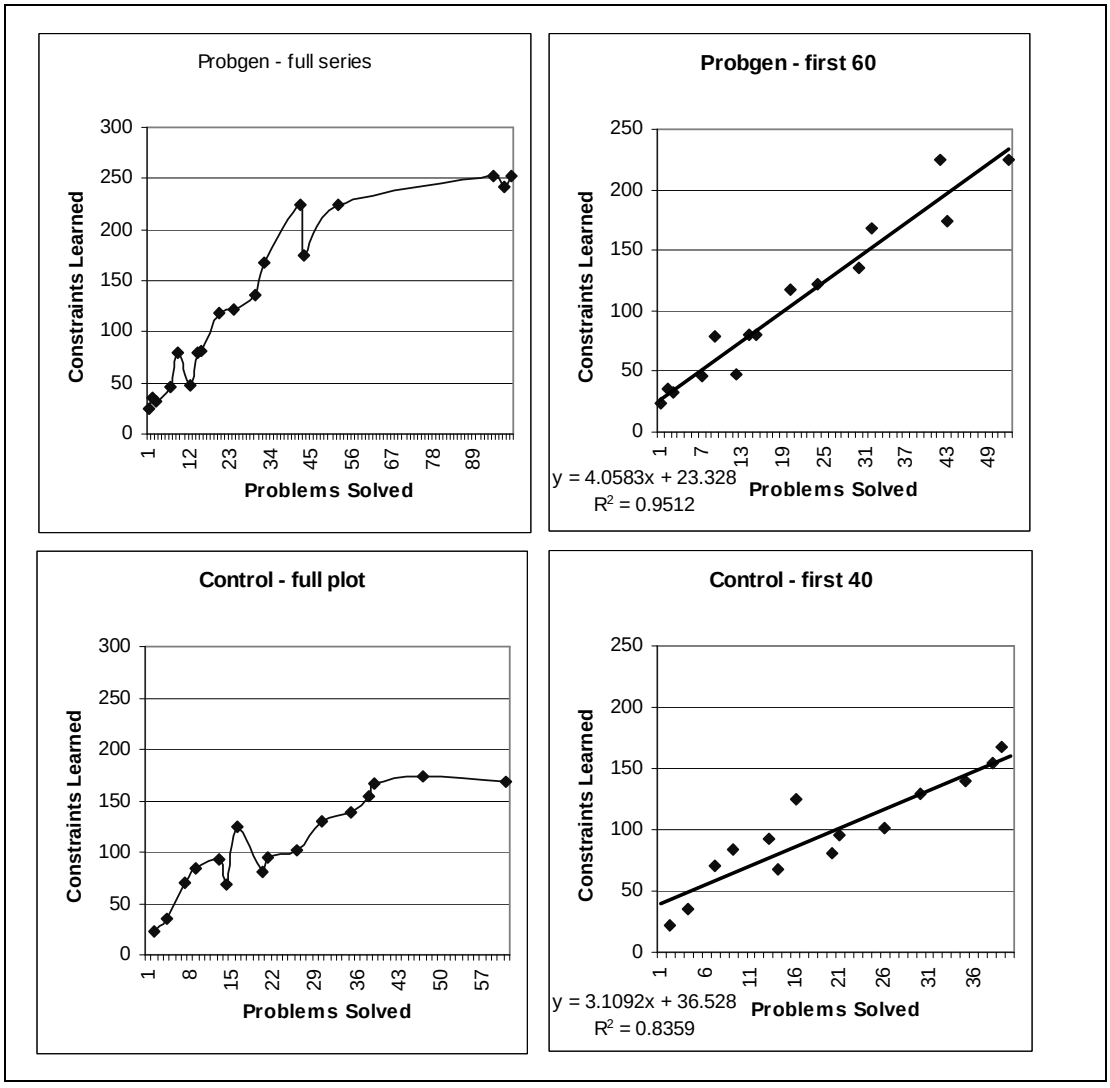

Figure 15. Constraints mastered per problem

6.9 Discussion

We presented four hypotheses for testing: two weak (6.1 and 6.3), which verify that Problem Generation works and does not degrade system performance, and two strong (6.2 and 6.4), which would suggest that Problem Generation is beneficial. We used static tests to gauge the practicality of Problem Generation, and found that even in the rudimentary version used for the evaluation (i.e. no natural language generation, errors/incompleteness in the instantiation constraints), the algorithm is effective and leads to a drastic reduction in the time taken to author new problems. We then obtained results from the classroom evaluation that showed no discernible detriment

in performance as a result of utilising the artificial problems. We propose, therefore, that hypotheses 6.1, 6.2 and 6.3 have been met.

The classroom analysis also showed that the rate at which learning occurs, i.e. the rate at which students reduce the number of errors they make per concept, was significantly higher when the generated problems were used. Efforts to find explanations for this effect, such as differences in the constraint sets, failed to explain the effect. Analysis of the difficulty of problems set also found no significant difference. We therefore conclude that the use of problem generation, coupled with the revised problem selection algorithm, leads to an increase in learning performance.

The generated problems themselves are unlikely to lead to such an effect. However, the fact that there are more problems and, in particular, that there is a larger number of more difficult problems, increases the likelihood that the new algorithm is able to find a problem of appropriate difficulty. Recall that the new algorithm measures the difficulty of each problem relevant to the student model. In the control group, the difficulty of each problem was a static value provided by the author. Problems may therefore be of appropriate structural difficulty for this student's aptitude, but be made up of inappropriate concepts. The effect of computing the conceptual difficulty is to raise the difficulty of the problem by some unknown amount depending on the concepts involved, and the student's grasp of them. Further, depending on the student's performance, it is possible that the control system might arrive in the position whereby there are no suitable problems because, when the conceptual difficulties are included, the problems all become too hard. Conversely, an advanced student may quickly exhaust a small problem set. The more problems there are covering many different subsets of the constraint set, the more likely one can be found for a given situation. Figure 16 illustrates the number of problems available for a given difficulty range.

It is clear that the increase in the number of problems leads to a better range of difficulties being available. For the authored problems, at a level above 300 (an average user) there are only one or two problems available at each level. For the generated problems, however, there are up to 20 problems at the 700 level (advanced users), with a trough in the middle. Although there is not an even spread of problems across all difficulties, there are at least a reasonable number of problems for advanced

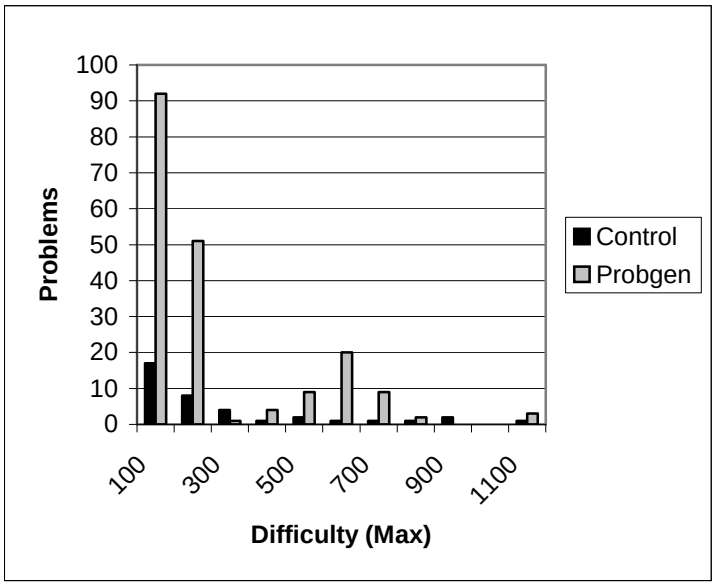

Figure 16. Problems available by difficulty

students. For the control group, in contrast, it is more likely that the system will run out of problems of appropriate difficulty. When it does so, it reverts to simpler problems, until all problems are solved. At this stage learning is probably negligible. To determine whether or not this occurred we analysed the individual logs to observe whether problem difficulty decreased below the student's proficiency level. For the 27 students in the control group who completed one or more problems, six of them ran out of problems, and the system began working backwards through simpler ones. In the worst case, the student completed 13 problems of lower difficulty because the system had no more difficult ones to offer. Further, six students exhausted the problem set completely. In contrast, for the experimental group, only four ran out of problems of suitable difficulty, and only after completing more than 80 problems. None of the students in the experimental group exhausted the entire problem set. This probably accounts for the graphs in Figure 15. The amount of learning that occurred once the system had run out of suitable problems is highly likely to have decreased.

We may also look at the number of problems per constraint, which gives an idea of how well the problem set covers the target domain. For the control group there is an average of 3 problems per constraint, with 20% of the constraint set being covered by one or more problems. In other words, on completion of all problems, the student will have covered 20% of the domain. For the experimental group, there are 7

problems per constraint, also covering 20% of the constraint set. Note that 100% coverage is not achievable, since around 50% of the constraints are syntactic, and are therefore dependant on how the answer is encoded, while approximately 50% of the remainder test for the *absence* of problems. We would therefore only expect around 25-30% coverage at most. These figures show that for each individual constraint (domain concept) there are twice as many problems available to cover them, so twice the likelihood that a problem can be found to teach a given constraint.

Finally, the two systems tested used a very different basis for selecting the next problem. The control group chooses a target constraint and selects the best problem for which that constraint is relevant. Problem Generation simply chooses the problem that best fits the model, regardless of what the previous concept being taught was. It is possible that the improvement observed comes from this change of tack: perhaps the "target constraint" method is too narrow and leads to excessive repetition. However, the effect of this method is more likely to be the reverse: because there may not be any more problems using this exact constraint, problem selection may often be uncontrolled. Whatever the reason for the observed improvement, the fact that problem generation allows a large number of problems to be authored quickly means that *any* problem selection algorithm will have more problems available to select from, which will allow it to more closely match problems to student models. We therefore submit that Hypothesis 6.4 is supported: that the problem generation and problem selection algorithms presented together lead to an increase in learning performance.

Recall, also, that the number of problems generated was limited to one per constraint to reduce the (initially unknown) effort required to vet the problems and translate them into English. Given how little time was required to perform this task, there is no reason why we could not have generated say, five problems per constraint, which would have made the problem set size even more favourable. Also, we were fairly ruthless in our elimination of unsuitable problems, throwing away 75 percent of the generated problems. Instead, we could have corrected these problems, which would have given a larger problem set. Finally, the instantiation constraints were fairly hastily arrived at, and were the chief cause of errors in the generated problems. Correcting these would have yielded a larger problem set. Therefore, the effects seen

here are merely an indication of how problem generation can improve an ITS: in practice the potential gains may be higher.

7 An authoring system for CBM tutors

The ultimate aim of any improvements to the efficiency of ITS production is to develop an authoring tool that permits rapid deployment of future systems with minimal duplication of work. There are two strategies for authoring: macro-level architectures, and micro-level tools. These approaches are complementary.

At the macro-level, a “black box” performs tutoring functions for multiple systems. All domain-specific information such as the name of the system, the knowledge base, the problem set and input-processing instructions are supplied as input to the system, which then presents problems and processes user input as required. This system may be in the form of an ITS *generator*, where the input is used to build a new tutoring system that runs as a separate executable, or an ITS *engine*, where the domain-specific information for multiple tutors is used to direct a single ITS process, which runs all the tutors in a single process. We have adopted the latter of these two approaches to develop a web-based ITS engine, which is further described in Section 7.2. This engine can serve multiple tutors over the internet, just as a single web server may serve up information from multiple sources.

At the micro-level, authoring tools are provided that allow a teacher to generate a new tutor with a minimum of effort. The problem generator described in Chapter 6 is an example of such a tool: given a domain knowledge base plus some additional information, the problem generator can produce an arbitrary set of problems that the author can then refine. Further, solution generation enables thorough testing of the constraint set: it generates many possible solution fragments that are considered valid with respect to one or more constraints, so may appear in student solutions. Finally, the new constraint representation has a heavily restricted syntax, which lends itself to a *constraint editor* to facilitate the writing of the knowledge base, or a *constraint inducer* that builds new constraints from examples of problems and solutions supplied

by the author. These tools are described in Section 7.3. Both the ITS engine and the authoring tools explore our final hypothesis:

> **Hypothesis 4**: Because the new representation is domain-independent, it may form the basis of an ITS authoring tool that supports the development of new CBM tutors.

7.1 Existing authoring systems

Many ITS authoring systems have been developed, using a variety of approaches. However, none have been built for CBM tutors. We describe some of the major achievements in ITS authoring.

7.1.1 REDEEM: adding instructional planning to CAI

The REDEEM authoring system (Major, Ainsworth and Wood 1997) is designed to allow teachers to build or customise their own computer-based coursework. Whereas conventional computer-aided instruction (CAI) generally presents educational material in a non-adaptive fashion, REDEEM allows individual teachers to overlay their strategies by categorising the material and describing key features about it such as familiarity, difficulty, generality, passiveness, questioning style and level of hinting. They may also add questions (and answers) associated with each page of material. They then describe the teaching strategy to be used for a given group of students such as level of student choice over presentation, teaching versus testing, generality of material to be chosen and hinting/feedback levels. All quantitative features (e.g. level of generality) are specified via a GUI interface by moving sliders to the appropriate position.

REDEEM also contains a tutoring shell, which presents the material to students using the information provided during the authoring phase. Students are assigned to one of the strategy groups, which determine how they should be taught. REDEEM also contains a pedagogical “black box”, which makes further fine-grained decisions, such as when to move from general to specific material, based on hard-coded rules derived from interviewing real teachers.

In a formative analysis (Ainsworth, Grimshaw and Underwood 1999) Ainsworth et al. found that teachers were able to easily tailor existing CAI material to their own teaching strategies. Variation between different teachers' strategies (plus the feedback given by the teachers) also indicated that being able to modify the strategies was worth the effort required. Two teachers with no previous experience of REDEEM or computer-based training completed a six to eight hour teaching session in around ten hours, much less than the 200 hours per hour of training estimated to create tutors from scratch (Woolf and Cunningham 1987). However, the time to create the coursework in the first place has not been considered. While prior coursework would benefit the creation of any ITS, in REDEEM the material can be used directly.

REDEEM was designed specifically for use by teachers to rapidly create or customise new courseware, and so is both easy to use and very general: any domain that can be taught using a storybook approach can be authored using REDEEM. However, the resulting system is shallow in that it does not contain a domain model with which it can provide detailed feedback or plan teaching operations (such as which problem to present next) to any fine degree.

7.1.2 Demonstr8: programming by demonstration

At the other end of the spectrum is Demonstr8 (Blessing 1997), an authoring tool for model tracing tutors. This system provides assistance in the creation of deep systems but for a limited domain set. It may currently be used only to generate arithmetic tutors, although Blessing claims the approach should be general enough to lend itself to other domain types. However, he says this would require the creation of new *authoring* systems. It aids tutor production at both the macro-level—by including the main components of a model-tracing tutor such as the model tracer, student model, and user interface—and at the micro level, by assisting the authoring of the underlying domain model. In Demonstr8, the author first uses GUI tools to define the interface using specialised widgets whose behaviour is domain-dependent. They then define the underlying declarative chunks or *working memory elements* (WMEs) by grouping together elements from the interface. For example, in a subtraction problem WMEs are created for each *column* of the problem/solution area by grouping together cells that are aligned vertically, and for *problems* by grouping together columns. Such

WMEs may be made directly from the interface components (e.g. grouping cells into columns) or by grouping other WMEs together (e.g. grouping columns into problems). The author must also define *knowledge functions*, i.e. functions relevant to the domain that the student would need to be able to perform. In subtraction for example, the student needs to be able to subtract two digits, so the tutor must also contain this function.

The most powerful part of Demonstr8 is the procedure-inducing tool. This uses programming by demonstration (Cypher 1993) to infer the procedural steps being carried out by the author as they demonstrate the solving of a problem in the domain: each time the author takes a step (i.e. changes a value in the interface), they either communicate to the tool how they did it (e.g. they add "5" to the rightmost column of a subtraction problem, by invoking the subtract function) or they simply carry out the step and leave the system to infer what they did, based on the WMEs and knowledge functions available. In Demonstr8 all actions are assumed to be the result of applying a knowledge function. If more than one function may have been applied, the author is asked to choose the correct one. Demonstr8 now builds a default production rule based on what it believes to be the conditions currently applying to the problem that are relevant to the step just taken, and the function used to take it. For example:

```
For the rightmost column C whose answer contains BLANK

If
    the top and bottom elements of C are applied to the SUBTRACT
    function
THEN
    the result can be placed in the answer field of column C.
```

By default, Demonstr8 applies the heuristic that the production being created for the current situation should be generalised in one dimension. For example, the procedure previously given may have been generated while subtracting numbers in column 3, yet the production generally applies to any column.

Many tasks require the modelling of subgoals. In subtraction for example, a subgoal may be the "carry procedure". In Demonstr8, it is up to the author to decide when to form a subgoal, and inform the system by providing the name of the subgoal. Authoring then proceeds as usual until the author indicates that the subgoal is

complete. A final task is to specify what the “skills” of the domain are for presentation by a skillometer. These are a high-level summary of the domain: each production is labelled according to what skill (or skills) is being utilised.

Once the procedures have been learned Demonstr8 now contains all the information necessary to function as a tutor in the specified domain. It includes a problem generator, which by default provides random numbers for problems. The generator can be constrained so that for example, only subtractions not requiring a “carry” are presented by ensuring the range of numbers available for the second row are always less than those for the top row. Demonstr8 provides a standard interface for the student in which they drag items (numbers) from a “palette” into the cells of the problem.

Within the context of arithmetic tutors, Demonstr8 has been shown to dramatically reduce the effort required to build a new tutor: 10 minutes versus many hours for a from-scratch implementation. However, this does not take into account the time spent building the authoring system and how many tutors would need to be built to recoup this effort. Although Blessing contends that the approach used in Demonstr8 could be broadened to other domains, the current system can only author a limited domain set. It contains many components that are specific to arithmetic domains, including the interface widgets, standard arithmetic knowledge functions (addition, subtraction, decrementing) and the problem generator. We do not know the effort required to build these, so are unable to judge whether it would have been quicker to simply author tutors directly in the arithmetic domain, perhaps building one and then copying and modifying it to produce others.

Finally, a considerable level of expertise (over and above domain knowledge) is required to build a tutor using Demonstr8. During the authoring session, many steps that may seem obvious to an expert in model tracing are not at all intuitive to domain experts. For example, how would an arithmetic teacher understand that they need to group cells into columns and problems in order that Demonstr8 can generate the necessary WMEs to represent the required procedures? In this regard many of the tools in Demonstr8 (including the WME generator) might be thought of as high level programming tools rather than end-user systems. A programmer is probably still required to build much of the system.

7.1.3 Teaching by simulation: RIDES

A very different type of tutor is based on simulation, where the student is given an artificial world in which they may carry out tasks in the chosen domain. RIDES (Munro, Johnson, Pizzini, Surmon, Towne and Wogulis 1997) is an authoring tool for automating the development of such tutors. As such, it falls somewhere between the extremes of REDEEM and Demonstr8: while the type of delivery is limited to simulation, the set of domains that may be taught in this way is more diverse than Demonstr8 (currently arithmetic), although not as broad as what may be taught by the story-book approach of REDEEM. Further, RIDES provides support for domain modelling, although the depth to which simulations are modelled is fairly low, hence it falls short of Demonstr8's ability to generate models to arbitrary complexity. As with both Demonstr8 and REDEEM, RIDES is both an authoring system and a shell: as well as providing help for generating the tutor, it runs the resulting system.

Authors generate tutors for procedural domains in RIDES by building a simulation of the procedure to be taught. RIDES provides a set of editors for creating the graphical components necessary to portray the domain and specifying how these objects behave. For example, a switch may have an attribute "State", whose value is toggled between the values "off" and "on" as the result of a mouse click. Similarly, a light may have a control "colour" which is set to "red" or "green" depending on the value of an attribute of another object (e.g. "green" if the "value" attribute of the "temperature gauge" object is 90 or less, otherwise "red"). The author then simulates the procedure to be learned by simply carrying it out, while RIDES records the actions taken.

RIDES automatically offers three modes of tutoring: demonstrate, practise and test. All three play back the simulation, but they vary in how this is controlled. In "demonstrate" mode, the simulation is simply played back verbatim, with the student's control being limited to pacing the display via mouse click. In "practise", the student is required to perform the necessary actions in response to the prompt "perform the next action". If they get it wrong, they are so informed and required to try again. After three attempts they are told what they should have done and the relevant item in the simulation is highlighted. In "test", RIDES behaves similarly to the previous mode except the student is immediately told whether or not each action is

correct, but are not told why they are wrong. RIDES records their actions and the state (right or wrong) of each.

RIDES greatly reduces the effort required to build tutors by heavily scaffolding the simulation authoring process, and automating the entire tutoring session. Sundry items such as text to be presented to the student before, during and after a procedure are created from "canned" text, such as "`you should have set <OBJECT> to <ATTRIBUTE-VALUE>`." Control of the session is also fixed, both at the procedural level (display initial text, step through the procedure, display final text) and at the session level. RIDES also automates the student modelling and presentation process: given a list of "objectives" and the mapping between objectives and procedures, RIDES decides which procedure to present next and when to move on to the next objective. However, the author may override many of these items using further editors to modify text, adjust the flow of a simulation, add new components to a simulation etc. Thus the authors of RIDES have overcome the dilemma of ease-of-use versus flexibility by providing two tools targeted at different audiences.

Like Demonstr8, RIDES uses a (basic) form of programming by demonstration to author the procedures. The main difference is that Demonstr8 tries to infer new rules based on incomplete information about why the user has carried out the step. Further, it tries to generalise the actions performed to other, similar actions. In contrast, RIDES simply records *exactly* what has been carried out and makes no inferences about it. Thus, whereas Demonstr8 tries to infer a deep, detailed model of the domain, RIDES creates models that by default are very shallow: in RIDES a particular step is necessary because *the teacher performed that step.* In contrast, an action in Demonstr8 may be applicable because the appropriate conditions have been met that make it valid to perform next.

In spite of the shallow modelling abilities of RIDES, it has been a very popular tool for simulation-based tutoring. This has been partly because the simulation tools themselves are so powerful that it has been integrated into other systems where simulation is required. It can also be used to generate tutoring *environments*, where the student is free to "play" in the domain and observe the consequences, rather than being required to follow a rigorous procedure. For example, a tutor for injection moulding gives the student the tools necessary to "create" a part, for which the system

then develops a mould to illustrates to the student the consequences of their design decisions (e.g. by combining two simple shapes into one, the mould now requires five parts whereas two separate moulds would only require two each). What is not clear is how well the intelligent tutoring parts of RIDES support learning.

7.1.4 Support for authoring the domain model

The domain model is generally the most difficult part of an ITS to build, so is a prime candidate for authoring aids. There are two major approaches: assisting the editing and visualisation of the model, and knowledge induction.

Demonstr8, described in the previous section, is an example of both approaches. It provides domain knowledge induction using programming by demonstration, with the output of the induction step being a default set of production rules for the actions taken. Demonstr8 also provides GUI editors for the creation of knowledge chunks (WMEs) and the creation/editing of production rules. In this system the user never directly modifies the code of the WMEs or production rules, but rather interacts with a dialog that is an abstraction of the underlying element. However, Demonstr8 still requires the author to identify working memory elements, decide how to use them to solve the problem and identify sub goals. Further, to date Demonstr8 has not been shown to be effective beyond the authoring of arithmetic tutors, nor is it obvious that the effort required to build the tool in the first place is justified. Importantly, it is not clear how it would fare for more complex domains.

Using a totally different approach, DNA (Shute, Torreano and Ross 1999) provides dialogues for extracting the important knowledge elements of a domain from a domain expert but does not encode it in any machine-useable way. A knowledge engineer is still required to encode the resulting domain model. However, DNA's approach may still be useful, since often the hardest part of developing a domain model is not deciding *how* to encode it, but rather *what* to encode. DNA makes explicit the kinds of knowledge required by defining the domain knowledge along three axes with associated dialogues for each. *Procedural* knowledge elements (PK) are lists of steps to be carried out, where each step can be further divided into sub-procedures analogous to goals and sub goals in ACT-R. *Symbolic* knowledge elements (SK) describe static facts, such as the definition of the term "mean" in

statistics. *Conceptual* knowledge elements (CK) describe relationships between SKs, such as how the mean relates to the shape of the underlying distribution. In a trial evaluation Shute et al. found that DNA allowed three people conversant in statistics (but not in ITS) to provide 62% of the knowledge elements needed for a statistics tutor (Stat Lady (Shute and Gawlick-Grendell 1993)) in only nine combined hours. A further aim of DNA is to produce a semantic network that captures the knowledge elicited, but for now this remains a major task.

Other authoring tools provide knowledge visualisation and editing functions. LEAP (Sparks, Dooley, Meiskey and Blumenthal 1999) builds systems that teach customer contact employees (CCEs) the skills for effectively responding to customer requests. The essential nature of a course unit is a dialogue between the CCE and the system (an artificial customer). LEAP allows for great variation and flexibility in how dialogues unfold. Authoring of such dialogues is supported by an array of GUI tools. A *Script Editor* provides the basic mechanism for developing a dialogue. In this editor the author creates each step in the dialogue, filling in the main attributes and leaving the rest to the system. The *Subdialogue Graph Editor* provides a graphical overview of the entire dialogue as dialogue nodes and transition nodes. Items may be added, deleted or expanded using this editor. The *Transition Editor* is for adding or modifying the details of a transition node, such as what response is required before the call can proceed from “discuss problem” to “determine problem”. The *Node Editor* and *Action Editor* are similar screens for entering the details of these components. LEAP thus provides a rich, multi-level means of editing and visualising the domain model, but does not help to induce its content. Other examples of this approach are IDE (Russell, Moran and Jordan 1988), Eon (Murray and Woolf 1992), and CREAM-Tools (Nkambou, Gauthier and Frasson 1996).

7.2 WETAS: A web-enabled CBM tutor authoring system

While CBM reduces the effort of building domain models for ITS, the task of building a new system is nevertheless still large. Several tutors we implemented in CBM share in common a textual user interface. To reduce the authoring effort, we have developed WETAS (Web-Enabled Tutor Authoring System), a web-based

tutoring engine that performs all of the common functions of text-based tutors. To demonstrate the flexibility of WETAS we have re-implemented SQL-Tutor (Mitrovic 1998), and developed a new ITS for teaching English Language skills (LBITS). Although these domains share the property of being text-based, they have very different problem/solution structures. WETAS is based on constraint-based modelling. It utilises the new constraint representation described in Chapter 4 to maximise the work performed by generic code. The architecture borrows heavily from SQLT-WEB, the web-based SQL-Tutor system (Mitrovic and Hausler 2000), with two main differences. First, the new constraint representation is utilised, along with a new constraint evaluator. This significantly reduces the amount of domain-dependent code in the solution evaluation part of the system, and cleanly separates the constraints from the evaluator. Second, a further "layer" of data input is added: as well as splitting the domain into problem subsets ("databases" in the case of SQLT-WEB), the system now further splits the overall tutoring information into domains. Thus, a problem presented to the student now belongs to a particular subset (e.g. database) of one of several domains. The constraint evaluation process has access to all of these things, so that constraints can be specific to individual problems (although, in practice, they never are), subsets (for example, in SQLT-WEB when testing for a valid attribute, the answer depends on which schema is currently active), and the domain being taught. The overall architecture is depicted in Figure 17.

7.2.1 Scope

(Murray 1999) divides systems for creating ITS into "authoring tools" and "shells". The former provide extensive aid in developing ITS, while the latter are merely a framework for building tutors, and so they support low-level tasks (such as interface development and data storage), while failing to decrease the effort involved in developing the "intelligent" aspects of the tutor. We consider WETAS to be an authoring tool (as well as a shell) because it provides many of the adaptive functions required of an ITS (problem selection, evaluation, feedback, student modelling, etc). It also provides custom representation for easily describing the problem set and the domain model. We now describe the scope of WETAS with regard to the four main

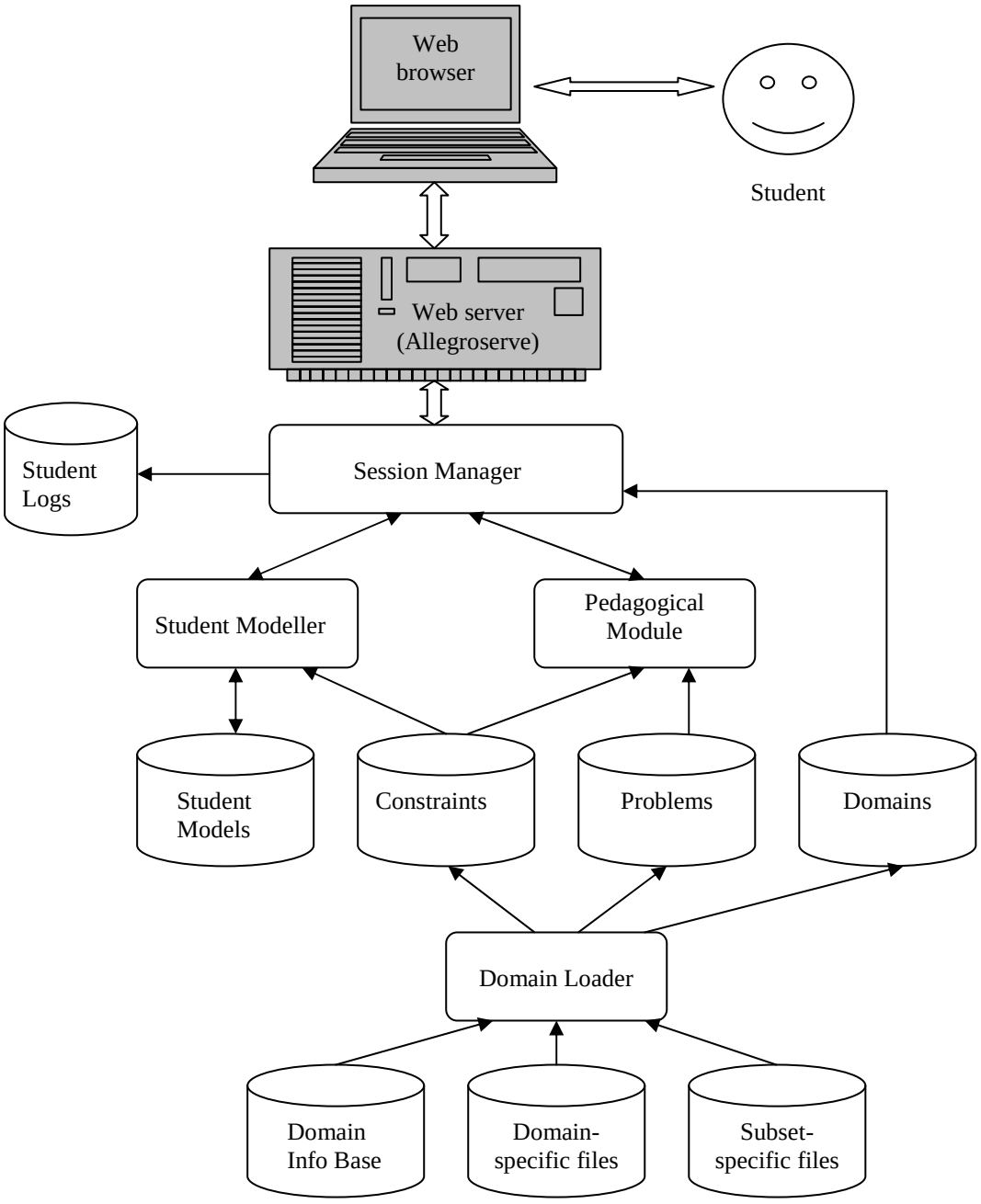

Figure 17. WETAS architecture

functions of an ITS: the student interface, domain model, pedagogical module and student model.

Student interface. WETAS completely automates the student interface. The layout is fixed, consisting of four panels: problem selection, problem/solution presentation,

scaffolding and feedback. Further, all but the scaffolding panel are driven automatically from the data.

Domain model. In WETAS authoring of the domain model is supported insofar as a language is provided for constructing the domain including macros for sub-rules. This language simplifies the creation of the domain model by removing the need to learn a complex programming language. WETAS also provides a generic domain modeller, in the form of the constraint evaluator. No further support is currently provided for writing the domain model, however we have undertaken some preliminary investigation into constraint editing and induction. This is discussed in Section 7.3.

Pedagogical module. Instructional planning in WETAS is fixed. All domains supported by WETAS are of the "learning by doing" kind. WETAS chooses the next problem to solve by evaluating the structural and conceptual difficulty (Brusilovsky 1992) of each candidate problem, and choosing the one that best fits the student's current knowledge state and level of ability. The problems themselves may be hand-written, or generated from the domain model using the algorithm described in Chapter 6.

Student model. Like most other authoring systems (Murray 1999), WETAS uses an overlay student model: each constraint includes a count of the number of times it has been relevant and how many times it has been violated, plus a trace of the behaviour of this constraint over the life of the model. The last four "hits" are used to decide whether the state of the constraint is currently "not learned" or "learned", with two successes in a row indicating that the constraint is learned. This information is used to calculate the conceptual difficulty of each problem, by increasing the difficulty by a constant amount for every relevant constraint that is not learned. Similarly, we increase the conceptual difficulty by another constant for every constraint relevant to this problem that has never previously been relevant. These constants are currently set to 5 and 10 respectively, i.e. a constraint that has been seen but not learned adds five times the difficulty to the problem as one that has been mastered, while a constraint that has never been seen adds ten times the difficulty. These constants were obtained empirically by using the system and observing which problems were selected. In

practise WETAS is not overly sensitive to these values. The difficulty each constraint adds to the problem is determined automatically by tallying up the number of terms in the constraint's match patterns, giving a measure of the effort required to complete the minimum parts of the solution necessary to satisfy this constraint.

When building ITS authoring systems, there is inevitability a trade-off made between flexibility (or generality) and depth (Murray 1999). The WETAS system supports deep tutoring by providing a robust constraint evaluator, student modelling functions and problem selection. It provides flexibility by supporting any domain where the problem and solution can be represented as (structured or unstructured) text. Further, it is possible to extend WETAS' capabilities to graphical domains, provided the problem and solution can be sufficiently described using text (see Section 7.2.9). The main trade-off is that WETAS does *not* currently provide flexibility of the student model and teaching strategy. However, the advantage of this is that the author is freed from such considerations. In the future we may modify the system to allow such components (or parts of them) to be provided by the author as "plug-ins", which is the case for scaffolding information now.

7.2.2 Implementation of WETAS

WETAS is a web-based tutoring engine that provides all of the domain-independent functions for text-based ITS. It is implemented as a web server, written in Lisp and using the Allegroserve Web server. WETAS supports students learning multiple domains at the same time; there is no limit to the number of domains it may accommodate. Students interact through a standard web browser such as Netscape or Internet Explorer. Figure 18 shows a screen from SQL-Tutor implemented in WETAS. The interface has four main components: the problem selection window (top), which presents the text of the problem, the solution window (middle), which accepts the students input, the scaffolding window (bottom), which provides general help about the domain, and the feedback window (right), which presents system feedback in response to the student's input.

WETAS performs as much of the implementation as possible in a generic fashion. In particular, it provides the following functions: problem selection, answer evaluation, student modelling, feedback, and the user interface. The author need only

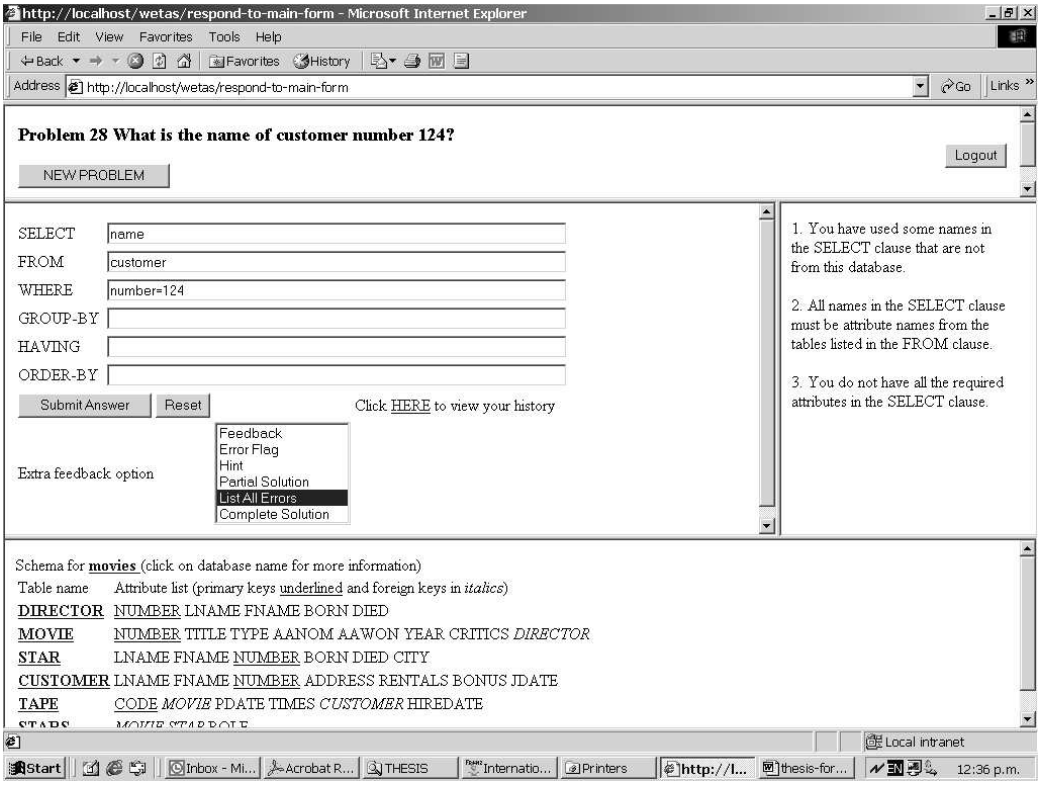

Figure 18. WETAS interface (SQL domain)

provide the domain-dependent components, namely the structure of the domain (e.g. any curriculum subsets), the domain model (in the form of constraints), the problem/solution set, the scaffolding information (if any) and, possibly, an input parser, if any specific pre-processing of the input is necessary. Each of these is now described.

The domain structure

All of the domain information in WETAS forms a hierarchy, where the top-level structure is the domain record. There is a domain record for each domain that the system supports. This record tells the system the name of the domain, the directory name where files relating to that domain may be found, where to find the scaffolding information for this domain, the name given to problem subsets, and the parser (if needed) for parsing the student's input prior to evaluation.

Exercises in each domain may be partitioned into *subsets*. For example, in SQL-Tutor the student may choose to answer questions that require queries to be written pertaining to one of several relational databases. Some information required by the system (including the problem set) is subset-specific, so each domain record includes

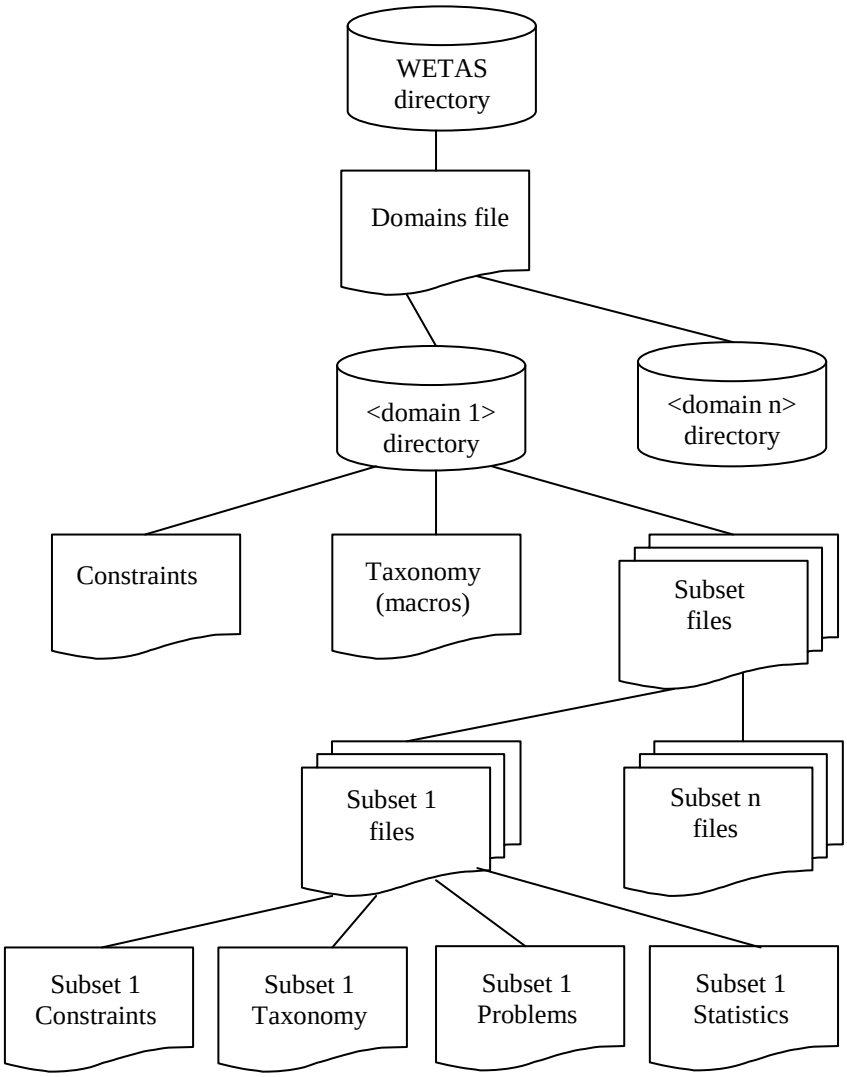

Figure 19. WETAS input files

a list of subset records containing this information. Also, the domain model may vary for each subset, so this is also stored at the subset level. Finally, each subset has its own list of problems. Figure 19 depicts the structure of the data input.

The domain model

The domain model is implemented as a modular set of constraints using the representation described in Chapter 4. Each domain may record constraints at two levels: those that are common to all subsets are stored at the domain level, while subset-specific constraints may also be provided. This allows the constraint set to vary

between subsets if needed without duplicating the common ones. For example, in the Language Builder domain, the puzzle “Rhyming Pairs” requires the answer to be two words that rhyme, as well as having the correct meaning. A constraint specific to this subset tests for rhyming pairs of words, while the words themselves and their meanings are stored at the domain level.

Many constraints require enumerations of the allowed values of a term in a match pattern. For example, a constraint in SQL that tests a table name is valid for the current database requires a list of all valid table names for that database. Further, some general concepts, such as “arithmetic symbol”, are also encoded by enumerating the list of valid values. Thus each domain requires a taxonomy that describes the atoms of the domain. However, some elements of the taxonomy are also subset dependent, such as “valid table” just described. The taxonomy is therefore also recorded both at the domain level (for domain-wide atoms such as “arithmetic symbol”) and at the subset level. The taxonomy is recorded as a set of macros, using the same representation as the constraints.

Problem Representation

As stated earlier, CBM critiques the student’s attempt by comparing it to an ideal solution. Each problem is therefore represented by the text of the problem plus the ideal solution. In WETAS problems and their solutions may be structured. In SQL-Tutor each problem consists of a text message describing the database query that must be written, while the solution consists of each of the six possible clauses of an SQL query (SELECT, FROM, WHERE, GROUP-BY, HAVING and ORDER-BY). In the Language Builder domain each problem consists of a set of clues, where the student must provide an answer for each clue (for example, they must type the plural version of each clue word). WETAS caters for different problem/solution structures by allowing a problem to have any number of *clauses*. Each clause nominally consists of the clause name, a text string that represents an ideal solution for that clause, an (optional) additional clue for that clause and the default input for that clause. However, the solution part of the clause may itself be a list of *sub clauses* again containing the sub clause name, ideal solution, a clue and the default field value. This

structuring may occur to any depth. In the Language Builder domain for example, nesting occurs to one level (see Section 7.2.5).

Scaffolding and parsing

Before a solution is fed to the constraint evaluator, it may require parsing to convert the text input into a list of words (or terms) that the pattern matcher can use. A default parser is provided, which splits text into words using white space and non-alphanumeric symbols as boundaries. However, some domains may have other parsing requirements. Each domain record contains a field that identifies the parser, which may be NULL (no parsing required), DEFAULT or the name of a LISP function that accepts the text input and returns the parsed result in a list. Similarly, domains may optionally provide scaffolding information. WETAS allows the author to specify either static HTML pages or dynamic functions.

WETAS has been implemented in prototype form and used to build two tutors to explore its capabilities and evaluate its effectiveness in reducing the ITS building effort.

7.2.3 Building an ITS using WETAS

Because WETAS is data driven, authoring a new ITS consists entirely of creating the data files needed to instruct it how to operate. The steps involved are:

1. **Create the domain record;**
    a. Decide upon the domain to be taught, and give it a name;
    b. Create the domain record (in domains.cl), including the definitions for any subsets;
    c. Create a directory that will hold all the files for this domain, as a subdirectory of the WETAS main directory.
2. **Create the problem set;**
    a. Decide how the problem will be presented, i.e. how it will be broken up. For SQL-Tutor, the exercises are split into the six clauses of a SELECT statement; for Language Builder, they are represented by repeated clues;

b. Create the file <subset-name>.probans for each subset, containing the problem definitions for that subset.

3. **Create the domain knowledge base;**
    a. Create the semantic and syntactic constraints that are valid for the entire domain, and the top level taxonomy (files constraints-semantic.cl, constraints-syntax.cl, and taxonomy.cl);
    b. Create any subset-specific constraints and taxonomies, if necessary (constraints-semantic-<subset-name>.cl, constraints-syntax-<subset-name.cl>, taxonomy-<subset-name>.cl).
4. **Create optional components;**
    a. Create a parser, if necessary;
    b. Create the scaffolding web page and/or functions, if necessary.
5. **Create the login page for this domain;**
6. **Run the newly created ITS.**
    a. Run “load-domains” to load the new domain;
    b. Restart the WETAS web server.

The main steps are now described in more detail.

1. Create the domain record

The file `WETAS/DOMAINS.CL` contains the definitions of each domain supported by WETAS. Each entry includes the domain short and long names, the scaffolding type (FILE, COMPUTED or NIL), the generic name given to subsets, the name of the parser (if any) and a list of all subsets. Each subset entry contains the subset long and short names, The size of the field(s) that will accept the answer and the default problem text. The latter is used when there is no specific textual problem statement for each exercise. In Language Builder for example, the problem is specified at a lower level by a series of clues, so the top-level problem statement is blank. The default problem statement is therefore used to provide a general message about solving the problem. Figure 20 shows a domain file with just a single domain

```
(setq *domains*
  '(
     ; domain record
     (
        "Language Builder"              ; long name
        "LBITS"                         ; short name
        NIL                             ; scaffolding type
        NIL                             ; scaffolding name
        "puzzle"                        ; what you call a subset
        NIL                             ; parser name

        ; subsets
        (
          ("Scrambled words - unscramble the letters to make a word
            that matches the clue"      ; long name
           "SCRAMBLED-WORDS"            ; short name
           20                           ; answer size
           "unscramble the letters in the brackets to make a word
            that matches the clue."     ; default problem text
          )

          ("Last Two Letters - each word begins with the last two
            letters of the answer before it."
           "LAST-TWO-LETTERS"
           20
           "Each word begins with the last two letters of the
            answer before it."
           )
        )
     )
   )
)
```

Figure 20. DOMAINS.CL

(Language Builder), which contains two subsets: “Scrambled words” and “Last two letters”. The comments indicate what each field represents.

2. Create the problem set

As described in 7.2.2, each solution is represented as a set of clauses where each clause may either be a single text string or a list of subclauses, which themselves can consist of further subclauses nested to any depth. Problem text can be attached at any level. In the two domains described, we have used fairly simple representations: SQL-Tutor uses a set of six text clauses, while Language Builder consists of a single clause—“clues”—for which there are a number of subclauses, together with a clue for each. Figure 21 shows problem entry 202 for the Language Builder domain, for the

```
(202
 NIL ; no top-level problem text
 (
   ("CLUES"
  ;    id  answer  clue                          default input
       ("1" "SHADE" "Out of the sun (5)"       "SH")
       ("2" "DEAL"  "Hand out the cards (4)")
       ("3" "ALIVE" "Not dead yet (5)")
       ("4" "VEIL"  "Cover (4)")
   )
 )
)
```

Figure 21. Example problem from LBITS/LAST-TWO-LETTERS.PROBANS

"last two letters" subset. Each numbered line is a separate clue. The first string in each clue is a number that identifies this clue. Next is the answer for this clue, e.g. "SHADE". The third string is the text of the clue itself, e.g. "Out of the sun (5)". The last string, which in this case is used only for the first clue, is used to initialise the answer field.

Although WETAS is design to accept free-form text, it is possible to use the structured nature of problem specifications to allow other types of interface. Consider the domain of Lewis diagrams in chemistry. The problem might be presented as a textual question (e.g. "what is the Lewis diagram for methanol?") where the student is required to draw the corresponding diagram. WETAS could do this by using the nesting ability of the problem specification to represent the problem solving interface as a grid of character fields, where the student enters the appropriate chemical elements and bond symbols. Figure 22 gives an example of such a problem statement. Each entry, labelled "1" through "5", is a line of a 7x5 grid. Each cell within this line (labelled "1" through "7") is a single cell in this row of the grid. Each cell is either empty or contains a symbol. Figure 23 illustrates how this would appear on the screen. Note that there is no requirement for the problem structure to be static across domains or subsets; each problem could be structured differently according to the needs of the question being asked.

3. Create the domain knowledge base

The knowledge base consists of the constraints for the domain, any subset-specific constraints and the taxonomies for the domain and subset. First, the pedagogically

```
(1
 "Draw the Lewis diagram for methanol."
 (
   ("DIAGRAM"
("1"
 (("1" " ")("2" " ")("3" "H")("4" " ")("5" " ")("6" " ")("7" " "))
)
("2"
 (("1" " ")("2" " ")("3" "|")("4" " ")("5" " ")("6" " ")("7" " "))
)
("3"
 (("1" "H")("2" "-")("3" "C")("4" "-")("5" "O")("6" "-")("7" "H"))
)
("4"
 (("1" " ")("2" " ")("3" "|")("4" " ")("5" " ")("6" " ")("7" " "))
)
("5"
 (("1" " ")("2" " ")("3" "H")("4" " ")("5" " ")("6" " ")("7" " "))
)
  )
 )
)
```

Figure 22. Example of a Lewis diagram problem

significant states are decided upon. Constraints fall into two broad categories—semantic and syntactic—and there is a file for each. Syntactic constraints are authored by deciding what are the important principles of constructing *any* solution in this domain. In SQL-Tutor these relate to syntax and grammar rules for constructing an SQL query. In Language Builder they are mostly related to spelling.

Figure 24 gives an example of syntactic constraint from each domain. In the constraint for Language Builder, the relevance condition first extracts the clause

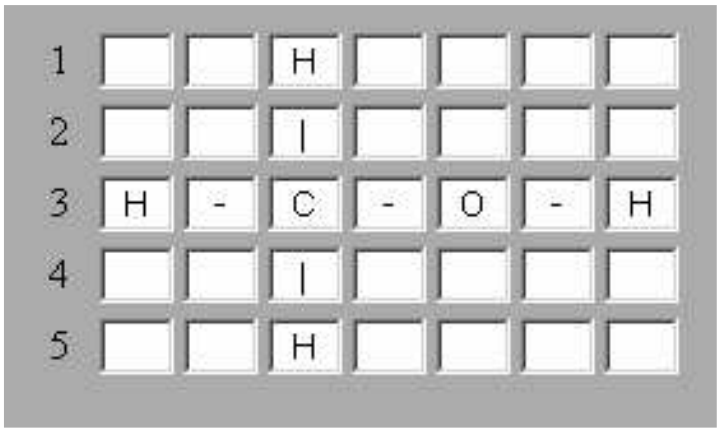

Figure 23. Screen appearance of Lewis diagram question

number and answer word from a clause in the student's answer. Then, it uses TEST_SYMBOL to test the letters within the word for "`i`" and "`e`", and binds `?letter` to the preceding letter. The relevance condition then checks that the preceding letter is not a "`c`". If it is, the constraint has been violated.

Semantic constraints relate the student's solution to the ideal solution in order to determine whether the question has been answered. They must be suitably flexible that they permit correct solutions that differ from the ideal solution. In SQL-Tutor the semantic constraints check that all of the necessary entities are present (tables, and attributes) and that they have been processed in the correct way, e.g. that conditions represent the same subset of records as those in the ideal solution. In Language Builder they check that the answers given have the same meaning as the clues. Figure 25 is an example of a semantic constraint for each domain.

The distinction between semantic and syntactic constraints can sometimes be blurred. In Language Builder constraints that test for appropriate letter groups in the answer are classed as syntactic because they are primarily checking that the word has been spelled correctly, yet they could also be called semantic since they are comparing the student and ideal solutions.

```
; syntactic constraint from SQL-Tutor
(61
 "A subquery in the HAVING clause must be enclosed within
brackets."

 (match SS HAVING (?* "SELECT" ?*))

 (match SS HAVING (?* "(" "SELECT" ?* "FROM" ?* ")" ?*))

 "HAVING"
)

; syntactic constraint from LBITS
(103
 "Remember: I before E except after C!"

 (and
   (match SS CLUES (?num ?* ?word ?*))
   (test_symbol SS ?word (?* ?letter "i" "e" ?*))
 )

 (not-p (test SS ("c" ?letter)))
 "CLUES"
)
```

Figure 24. Examples of syntactic constraints

```
; semantic constraint from SQL-Tutor
(55
   "You do not need all the tables you used in FROM."

  (and (not-p (match SS WHERE (?* "SELECT" ?*)))
       (or-p (match SS FROM (?* (^name ?t) ?* "ON" ?*))
             (and
                (not-p (match SS FROM (?* "ON" ?*)))
                (match SS FROM (?* (^name ?t) ?*))
             )
       )
  )

  (or-p
     (match IS WHERE (?* "FROM" ?* ?t ?*))
     (match IS FROM (?* ?t ?*)))

     "FROM")

; semantic constraint from LBITS
(2002
   "The words 'wear', 'ware', 'were' and 'where' mean different
    things. Have you used the right one?"

  (and
    (match IS CLUES (?num ?word1 ?*))
    (match SS CLUES (?num ?word2 ?*))
    (test IS (("wear" "ware" "were" "where") ?word1))
    (test SS (("wear" "ware" "were" "where") ?word2))
  )

 (test SS ((?word1) ?word2))
 "CLUES")
```

Figure 25. Examples of semantic constraints

4. Create optional components

If the domain requires any special parsing of the input (e.g. SQL parses TABLE.ATTR into the list `("TABLE" "." "ATTRIBUTE")`), a custom parser must be written. It may be either written in LISP, or callable from the LISP code. The standard parser, which splits a text string by white space and symbols, can be used as a guide.

Scaffolding information may also be provided. This can be either a collection of HTML documents or a function. For the former, the author provides a list of filenames to be published as URLs, where the file relating to the first member of the

list will be displayed in the scaffolding window, and the other members are assumed to be linked to it. If the scaffolding is provided by a function, this must be written in LISP or callable from the Allegroserve server.

5. Create the login page

Each domain has its own HTML login page. Any format is acceptable provided it passes the domain name, student login name and student difficulty level to the server by posting the URL WETAS/LOGIN. The login page must be located in the domain subdirectory and be named LOGIN.HTML. A template page is provided.

6. Run the new ITS

WETAS has now been provided with everything it needs to tutor in the new domain. The function LOAD-DOMAINS reads the domain file, including the new domain entry, and loads all the other files associated with each domain. It builds a domain entry in memory containing all the information from the domain file plus the problem and constraint sets for each domain. It also calculates the problem selection statistics for all domains by calculating the structural difficulty of each problem and the conceptual difficulty that would be added by each constraint. Finally, the WETAS server is restarted and the new domain is published along with all existing domains. The new ITS is ready for use.

We now describe two domains that we have implemented in WETAS.

7.2.4 Example domain 1: SQL-Tutor

SQL-Tutor (Mitrovic 1998) teaches the SQL database query language using Constraint-Based Modelling. It is available to the general public on the web (http://ictg.cosc.canterbury.ac.nz:8000/sqlt-web-login), and is used at Canterbury University in second and third year database courses. Students are given a textual representation of a database query that they must perform and a set of input fields (one per SQL clause) where they must write an appropriate query. This system was implemented in 1998 as a standalone tutor, in 1999 as a Web-enabled tutor, and has been re-implemented in WETAS.

Figure 18 (Section 7.2.2) shows a screen shot of WETAS running SQL-Tutor. When WETAS is first run it loads the domain information for all supported domains,

including the domain model and the set of problems to present. The student first logs on via a hard-coded HTML page that is specific to this domain. Once the student has entered their username and submitted the form, WETAS creates (if this is the first time the user has used this domain) or loads their student model and generates a student record stating which domain this student is currently using. After logging on, the user may select one of available databases on which they can practise queries; each database is a separate *subset* as described in Section 7.2.2. The student may change subsets at any time. WETAS stores a separate student model for each domain that this student is studying, and the current subset (i.e. database) is stored in the student model. The initial logon page is one of the few domain-specific parts of the WETAS system: nearly all functions are generic and data-driven from the student model, domain model and problem sets.

WETAS then selects a problem using the method described in Section 7.2.2 and presents it to the student. They then enter their solution and submit it for evaluation. The solution is first passed to an SQL-specific parser, which separates the input text into words. It then post-parses any qualified names (i.e. `TABLE.FIELD`) into LISP lists (i.e. `(TABLE "." FIELD)`) so that the constraints may test the individual parts of the name. The constraint evaluator compares the solution to the ideal solution using the constraint set for this domain and subset. In the SQL-Tutor domain there are no subset (i.e. database)-specific constraints as such, however around half of the macros (such as “valid_table” and “attribute_of”) are database-specific. Based on the results of this evaluation the feedback panel then conveys appropriate feedback, such as a success message, a list of error messages (obtained directly from the violated constraints) or the correct solution to the problem. The feedback types provided are the same as the original SQL-Tutor (see Section 2.4.5).

WETAS provides two mechanisms for scaffolding information: the author may provide either an HTML page or a LISP function that generates the information dynamically. In SQL-Tutor the latter is used to provide multiple levels for information about the database, from a description of each table to detailed help about field data types.

We have successfully reimplemented SQL-Tutor in WETAS with no difficulties arising. The only domain-specific parts of the system are the constraints, the problem

set, the login page, the scaffolding information and the parser. Of these, only the constraints may be considered an "intelligent" component. Thus, the author is freed to concentrate on the most complex part, namely development of the domain model.

7.2.5 Example domain 2: Language Builder ITS (LBITS)

Language Builder is an existing paper-based teaching aid that is currently being converted to a computer system. It teaches basic English language skills to elementary and secondary school students by presenting them with a series of "puzzles" such as crosswords, synonyms, rhyming words and plurals. For a subset of these puzzles, the general form is that of a set of clues where the student must perform some action on each clue to obtain the result, e.g. provide a word starting with "bl" that matches the meaning of the clue or provide the plural of the clue word. Figure 26 shows LBITS in action.

We created an ITS from Language Builder (LBITS) by adding a domain model so that feedback could be expanded from a simple right/wrong answer to more detailed information about what is wrong, such as that the meaning of their answer didn't match the meaning of the clue or they have got the letters "i" and "e" reversed. No special parser was required for this domain, nor was any scaffolding information

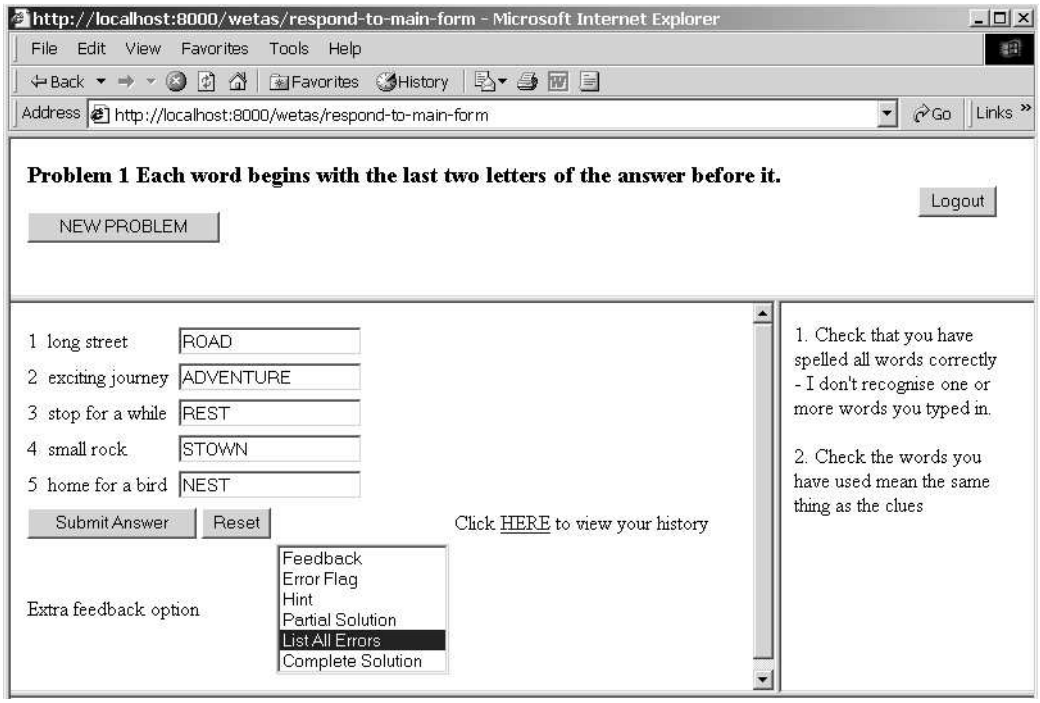

Figure 26. WETAS running the Language Builder (LBITS) domain

needed. Since the problems were already provided in paper form, the authoring task was limited to producing instantiations of the problems, encoding them in a suitable form and writing the constraints that form the domain model. For the latter we used a standard school spelling reference book (Clutterbuck 1990). Most of the constraints came directly from this resource book. For example, Clutterbuck groups words by letter groupings, such as those containing “able”. For each group we wrote a constraint that tests that if the ideal solution contains this pattern of letters, so does the student’s answer. Other constraints checked for commonly confused homonyms, such as “lose” and “loose”. We then added a few general constraints, such as one for each letter of the alphabet, to check the student had not missed any letters.

A problem consists of a list of clues, each requiring a word to be filled in. To achieve this, we took advantage of the ability to nest structures, as described in Section 7.2.2. For example, the problem specification for the exercise being solved in Figure 26 is:

```
(1
   ; IS - (#  answer   clue  default-input)
   (  ("CLUES"
            ("1" "road"      "long street"      "ro")
            ("2" "adventure" "exciting journey" "")
            ("3" "rest"      "stop for a while" "")
            ("4" "stone"     "small rock"       "")
            ("5" "nest"      "home for a bird"  "")
      )
    )
)
```

In this puzzle the user must enter a word that has the same meaning as the clue, where the first two letters of each answer is the same as the last two letters of the previous word. There is only one clause (“CLUES”), but this clause, instead of having a single text answer (as is the case for SQL), consists of a set of clues, each with their model answer and the default value for the solution. WETAS thus presents this structure as a table of clues with one entry field per clue for the answer.

Language Builder includes other puzzles, however these are graphical in nature, and are currently beyond the scope of WETAS (see section 7.2.8). The puzzles we have so far implemented are:

1. **Scrambled Words.** The student is presented with a set of letters and a clue. They must use the clue to build a word from the letters;

2. **Last two letters.** For each clue, think of a word that has the same meaning, where the first two letters of the word are the same as the LAST two of the answer to the previous clue;

3. **Plurals.** Produce the plural of each clue word, e.g. “oxen” for “ox;

4. **Rhyming word pairs.** Given a clue phrase, produce a pair of rhyming words that have the same meaning, e.g. for “beautiful energy”, an answer is “flower power”.

For the evaluation, we authored problems for the first two types of puzzle: “Scrambled letters”, and “last two letters”. For “scrambled words” the problems were created by calculating the structural difficulty of each word using the algorithm described in Section 6.4. The words were then sorted by difficulty and grouped into sets of around five, each of which forms a single problem, giving a total of 200 problems. A clue was then written for each word. For “last two letters” we used generated sets of (up to six) words that met the “last two letters” rule plus an additional rule that no words be repeated. This yielded 22 problems.

LBITS makes use the hierarchical nature of constraints but not of the taxonomy, since the “world” from which answers may be drawn is the same for all puzzles, i.e. an English vocabulary suitable for the target audience. Examples of subset-dependant constraints are: in “Rhyming word pairs” each pair must rhyme; in “scrambled words” each word must use the letters provided; in “last two letters” each word must begin with the last two letters of the previous answer. The system consists of between 20 and 200 problems per puzzle and a total of 315 constraints.

7.2.6 Evaluation

To determine how WETAS supports ITS building we rebuilt SQL-Tutor and built the Language Builder ITS. We tested LBITS in an elementary school classroom of nine children aged 11 and 12 from Akaroa Area School, to evaluate whether or not it was an effective learning tool. This trial was formative only: we were interested in what the students attitude was towards the system and whether or not their performance indicated that learning took place during the trial. To test the system subjectively we requested that each student fill out a questionnaire at the conclusion of an initial one

Log	Attempts	Problems completed	Attempts/ Problem	Final score
1	32	11	2.6	860
2	60	12	4.7	860
3	19	6	2	920
4	1	1	1	600
5	26	6	3.7	620
6	3	0	N/A	440
7	37	7	4.4	680
8	44	12	3.6	920
9	35	9	3.3	680
Average	28.6 (17.8)	7.1 (4.2)	3.2 (1. 2)	731 (158)

Table 10. Summary data for the LBITS evaluation

hour evaluation session (see Appendix B). At the end of the evaluation we plotted the constraint error rates for the group, in the hope of attaining the expected "power curve". Table 10 summarises the evaluation session. "Attempts" is the total number of attempts made to solve a problem during the 50 minute session. "Problems completed" is the number of problems the student answered correctly, irrespective of whether or not they required help. "Attempts/problem" is the number of attempts for each solved problem (i.e. excluding attempts for the last problem, which they abandoned at the conclusion of the session). "Final score" lists the difficulty rating for each student at the end of the session. The last row lists the averages of these figures, with standard deviations in parentheses. The nine students solved an average of just over seven problems each, (SD=4.2), taking an average of 3.2 attempts per problem. Two students (4 and 6) performed much worse than the others, while students 1 and 3 seemed to find the problems the easiest. This corroborates with observations during the session.

The students were very positive towards the LBITS tutor. Table 11 summarises their responses to the survey. Note that the first columns do not add up to nine because some participants ticked more than one box. Column one shows which

Which Puzzle	Difficulty	Ease of use	Enjoyable?	Learned?
Scrambled: 9	Too easy: 2	Easy: 9	Fun: 8	A lot: 7
Last two: 2	About right: 8	Okay: 0	OK: 1	A little: 2
	Too hard: 2	Hard: 0	No: 0	None: 0

Table 11. LBITS survey results

problems the students attempted ("scrambled words" or "last two letters"). The second column indicates how difficult they found the problems (one student ticked all three boxes, while another ticked both "too easy" and "two hard", to indicate that some problems were too simple and others too difficult). Columns three and four indicate how easy they found the interface to use and whether they thought using the system was fun. The last column indicates how much they thought they learned. These results indicate that on the whole the students enjoyed using the system, felt that the difficulty of the problems was about right and felt they had learned a substantial amount. All of them found the interface easy to use. Note that it is not possible to determine the relationship between performance (table 10) and subjective evaluation (table 11) because there was no way to identify which participant was which.

We plotted the probability of failing a given constraint as a function of the number of problems attempted for which this constraint is relevant, in the same manner as described in Section 6.8.5. Figure 27 shows the result obtained. It suggests that no learning took place. However, a number of the constraints arguably do not represent principles of the domain. Constraint 9000 checks that the student has filled in an answer, yet their failure to do so is most likely because they do not know the answer, rather than because they did not realise that one was necessary. It therefore does not

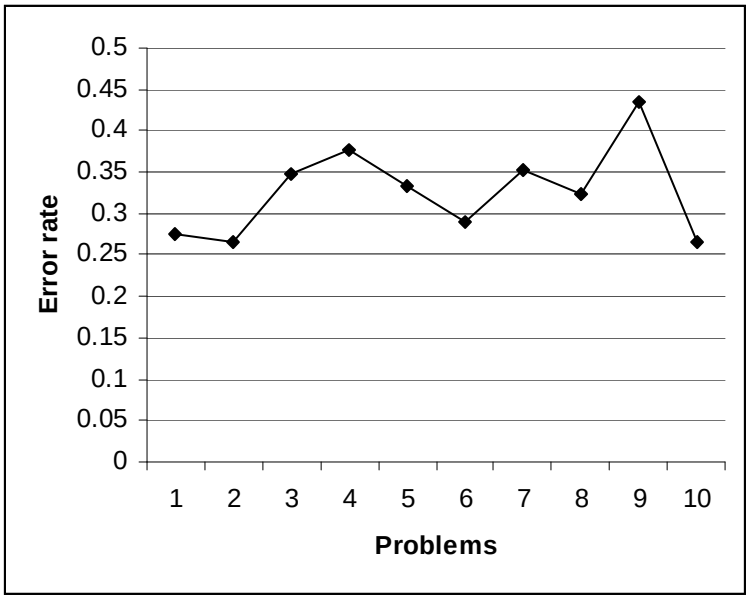

Figure 27. Error rate for raw constraint data

represent a knowledge structure that the student is trying to learn. Similarly, constraints 201 to 226 test that each letter of the alphabet is present if it is required. Again, these constraints will be failed if the student fails to fill in the answer, yet this is probably because they failed to recognise the required word *as a whole*, rather than because they failed to notice that this particular letter is required. In other words, the situations in which a student failed to fill in a particular are probably not pedagogically equivalent, which is a fundamental requirement of constraints. This is particularly true for "scrambled words" because students are given the letters as part of the clue. In contrast, if a student fails to recognise the required word from the letters provided, it is possibly because they are weak on words of that form, which *are* represented by the constraints that test for common letter patterns, such as "ough". We tested this by removing constraints 9000 and 201 to 226, and plotting the error curve again. Figure 28 shows the result. We now see the familiar "power curve", with a good degree of fit ($R^2 = 0.83$). This suggests that the students learned the domain with respect to these constraints during the session. Note that, as described in section 6.8.5, the power curve degrades as the number of attempts increases, because of the decrease in data volume. The graph in figure 28 is cut off at the point where the power curve fit is maximal.

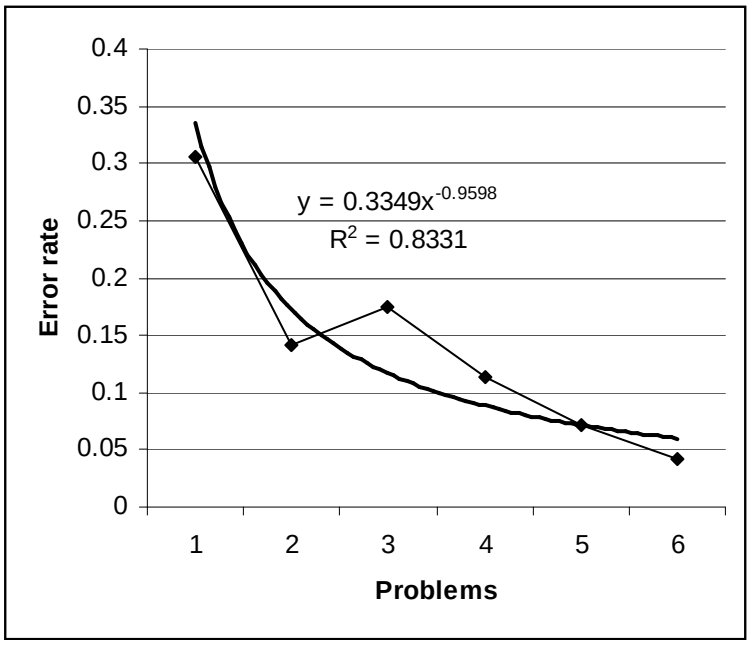

Figure 28. Error rate for revised constraint set

7.2.7 Conclusions

Constraint-Based Modelling (CBM) is an effective approach that simplifies the building of domain and student models. We have developed a prototype authoring system called WETAS for web-enabled tutors using constraint-based modelling, which we intend to use to develop further tutors for continued research into CBM and for release into classrooms. Of the two tutors built using WETAS so far, SQL-Tutor is a mature ITS that has been used in the classroom for two years and will continue to do so. Language Builder has been implemented in prototype form and evaluated on elementary school students. The evaluation demonstrated that LBITS, a system that was built in a very short time, is a usable, effective ITS. The reimplementation of SQL-Tutor under WETAS was straightforward, and the conversion of Language Builder from a paper-based instructional system to a full ITS has been similarly efficient, with the only major effort being the construction of the domain model. However, even this later task is made easier by the simple pattern matching language. By building an effective authoring tool using the constraint representation introduced in Chapter 4, we have satisfied Hypothesis 4.

WETAS draws upon the strengths of CBM, plus research carried out to date in practical implementations of CBM. It appears to be a promising tool for easier development of new tutors and a useful step towards the large-scale deployment of Intelligent Tutoring Systems.

7.2.8 Further work

The problem set used in any tutor supported by WETAS is currently static. We have developed an algorithm for generating new problem sets automatically from the domain model, however this is currently an offline process that requires human intervention. We are investigating expanding this algorithm to be able to generate new problems from the *student* model on the fly. Such an approach would allow the fit between problem selection and student knowledge to be controlled to a very fine degree. The major obstacle is the potential for the generated problems and their solutions to be incorrect or semantically unsuitable because of errors or incompleteness in the domain model. However, this approach is practical for simpler domains, such as LBITS.

When authoring an ITS the most difficult task is creating the domain knowledge base. Although the modular nature of constraints reduces the complexity of the model and our constraint representation simplifies the mechanics of encoding the constraints, nevertheless the complex task of determining what should be in the model and how to represent it remains. We discuss two possible solutions in Section 7.3.

WETAS currently only supports text-based problem solving. However, this limitation is only imposed because of the standardised user interface: as long as it is possible for the requirements of the problem and solution to be represented using text, the CBM approach is still valid (e.g. KERMIT, a CBM-based ITS for entity-relationship modelling (Suraweera and Mitrovic 2001)). We would like to extend WETAS to include tutors with graphical interfaces, possibly via plug-in interface modules.

Finally, the student model and teaching strategy in WETAS are fixed. We would like to be able to specify these as “plug-ins” such that different strategies might be tried and, more importantly, that individual domains may have different strategies and associated models.

7.2.9 Other domain paradigms

The domains described so far are all text-based, which is a limitation of the user interface. The constraint representation has no such limitation: all that is needed is an appropriate representation of the problem and solution such that constraints may be written that can critique the student's answer.

Consider an ITS for database design, such as KERMIT (Suraweera and Mitrovic 2001). The problem is set as a textual description and the answer entered as a diagram, from which key information is used to populate structures that represent each graphical object, such as entities and relationships. To be included in WETAS, some form of graphical editing facility would need to be provided, which is beyond the scope of what is discussed here. However, once the solution has been obtained it is a simple matter to convert the resulting data structures into strings, where each string represents some facet of the solution (e.g. "ENTITIES") and the values consist of an identifier followed by the various data fields from the original structure. The constraints can then test against these field values by matching against some sort of delineator, the identifier and the field value being tested. For example, consider the following (abridged) constraint from KERMIT, which checks that all entity and relationship names are unique:

```
Id         = 10
RelCond   = "t"
SatCond   = "unique(join (SSE, SSR))"
Feedback  = "Check the names of your entities and relationships.
             They must be unique"
```

Suppose that entities are represented by a clause "ENTITIES", and relationships similarly in "RELATIONSHIPS". The above constraint could now be encoded as in Figure 29.

7.3 Prospective authoring tools

The authoring system described so far focuses on the delivery of intelligent tutoring, and provides a substantial framework for authoring new domains. The other approach, as already mentioned, is to provide micro tools for building the various components. We now describe two possible additions to the authoring system.

```
(10
"Check the names of your entities and relationships. They
            must be unique"
(or-p
   (match SS ENTITIES (?* ?name ?*))
   (match SS RELATIONSHIPS (?* ?name ?*))
)

(or-p
   (and (match SS ENTITIES (?* ?name ?*))
        (not-p (match SS ENTITIES (?* ?name ?* ?name ?*)))
        (not-p (match SS RELATIONSHIPS (?* ?name ?*)))
   )
   (and (match SS RELATIONSHIPS (?* ?name ?*))
        (not-p (match SS RELATIONSHIPS (?* ?name ?* ?name ?*)))
        (not-p (match SS ENTITIES (?* ?name ?*)))
   )
"ENTITIES-RELATIONSHIPS"
)
```

Figure 29. Constraint for checking uniqueness of relationship names

7.3.1 Constraint learner

Although encoding an individual constraint is relatively straightforward, building an entire domain model is difficult and time-consuming. There are two main problems that can arise. First, the model may be missing one or more constraints and so the set of solutions that it describes will include some that are incorrect. When adding new problems, it is hard to evaluate the existing model to determine which necessary constraints already exist and which need to be added. We might achieve this by trying incorrect solutions and seeing if the constraint set detects the error. However, this is a very time-consuming task. Also, since the set of incorrect solutions to a given problem is large (if not infinite), it is highly unlikely that all potential (or even likely) problems will be found.

Second, an existing constraint may be too specific and so a valid answer to a new problem may be rejected. This too is difficult to detect. For example, the most obvious French translation to an English sentence may be accepted but an equally acceptable alternative rejected because a constraint has been too specific in detailing the equivalent meaning in French.

The modularity of constraints makes it possible to add new constraints individually provided that they aren't duplicated. Each constraint is a "truth" in its own right. This property suggests that it might be practical to learn constraints

automatically. A machine learning tool for constraint acquisition might provide three of the types of assistance that an authoring tool can offer: (Murray 1997)

1. Make knowledge/data entry more efficient;
2. Help the author articulate implicit knowledge, and;
3. May create new knowledge beyond what the expert might know or deduce.

We now describe an application of machine learning to constraint model acquisition.

Automating the Acquisition of Constraints

A constraint is a generalised form of a problem and solution. We could trivially build a domain model that consists of a set of patterns that represent problems and their (single) solutions, for example:

```
("You have translated the sentence incorrectly. Please try again."
         (MATCH PROBLEM "My name is Suky")
         (MATCH SOLUTION "Je m'appelle Suky")
)
```

Such a model is not very helpful because it cannot give the student any reasons for mistakes. Furthermore, it will reject alternative answers. One solution is to generalise each problem/solution pair such that it tests some underlying concept of the domain that is relevant to this problem. For example, we could generalise the previous example by ignoring all but the proper noun and replacing the value of the proper noun with a test for nouns, giving:

```
("You have used a noun which is not specified in the problem."
    (MATCH SS (?* (!noun ?n) ?*)
    (MATCH IS (?* ?n ?*)
)
```

Assuming that some of the underlying concepts such as "noun" already exist, we could automate the process of generalising the "trivial" constraint represented by the problem and solution by systematically generalising combinations of terms. Each potential constraint could then be compared against the existing set to ensure it is not a duplicate of an existing constraint. Finally, all possible problem/solution pairs could be generated which satisfy the constraint, with a human teacher evaluating them to determine whether or not they are valid.

Unfortunately this approach is vastly inefficient. First, the potential search space is immense. In the previous simple example, we would need to test a problem/solution pair for every proper noun, which could easily be dozens, if not more. The more generalised terms in the constraint, the larger the set of combinations is. Second, there will be many duplicate candidate constraints generated using this method, which will then need to be tested against the existing set and deleted. To overcome these problems, we look to the field of machine learning.

Learning by Asking Questions

MARVIN (Sammut and Banerji 1986) is a machine learning system that learns new concepts by generalising examples using existing knowledge. A teacher provides MARVIN with an example of each concept to be learned. MARVIN then uses its existing knowledge to generalise the example and produce a new trial concept. It then tests the trial by constructing an example, which is presented to the teacher. If the example is negative, the concept is incorrect and is either specialised or discarded. Conversely if the example was correct, the trial concept is still consistent and is subjected to further generalisation. Once this process has been exhausted the new concept is complete. It is then named (by the teacher) and added to MARVIN's long-term memory. Other concepts may now be built which include this latest one.

This approach is similar to that described in the previous section. However, MARVIN uses some heuristics to guide the search for the concept. First, MARVIN is able to specialise as well as generalise. If only generalisation were possible, MARVIN would only be able to make new trials involving a single extra concept, since it generalises one aspect at a time. However, there may be cases where a term in the example can be generalised provided that a conjunction of one or more already known concepts is true. This allows it to make generalisations that would otherwise be impossible. Specialisation is performed by looking for further concepts that match at least one of the original problem predicates that was discarded in the generalisation process. This heuristic efficiently searches the set of possible conjunctions of concepts to find those that are most likely to yield a consistent new one.

The second heuristic caters for the problem of enumerating and testing all possible instances of the new concept. MARVIN creates just a single *critical example*, which

has a high likelihood of failing if the concept is inconsistent. To do so, it creates the list of all elaborations of the initial example, i.e. the list of all known predicates which are true for the example. Next, it does the same for the trial concept. The critical example is then built, which is a valid example of the trial concept, but which does not satisfy any predicates for the initial example which are *not* valid for the current trial concept, i.e. it does not satisfy any conditions of the set

(All_Elaborations(Example) – All_elaborations(Trial))

Any such example has a high likelihood of being incorrect if the trial concept is inconsistent. For example, suppose we are trying to learn the concept “stackable”, where stackable objects are blocks. We have a simple domain theory, and present a single example of a stackable object as follows:

```
Domain Theory:
          Any_shape(X) :- rectangle(X).
          Any_shape(X) :- square(X).
          Any_shape(X) :- triangle(X).

          Block(X) :- rectangle(X).
          Block(X) :- square(X).

Stackable example: rectangle(A).
```

MARVIN first elaborates the example, by building a list of all matching predicates, giving:

All_elaborations(Example) = {rectangle(A), block(A), any_shape(A).

The most general possible trial concept is any_shape(A). This is now elaborated, however no other predicates apply, so the elaboration set is just:

All_elaborations(Trial) = {any_shape(A)}.

Finally, MARVIN creates a crucial example, i.e. an example of the new trial that is *not* a member of {rectangle(A), block(A)}. The only possible example is

“triangle(A)”. MARVIN presents this example to the teacher, who rejects it. MARVIN now repeats the exercise for the trial concept “block(A)”, and presents “square(A)”. This is accepted, so the concept “stackable(X) :- block(X)” has been learned.

In MARVIN this algorithm is performed recursively for each new example given to it, and so each example generates a single new concept. In the case of learning constraints, each example may embody multiple constraints. The task therefore is to construct as many valid concepts as possible from each example. Some parts of the example will be superfluous to each concept to be learned, while others will be critical. Further, whereas MARVIN can add the newly learned concept to the domain theory and use it to generate others, a constraint is the end of the line: constraint-based models are non-hierarchical. Finally, MARVIN uses first order predicate logic to define concepts, whereas we use pattern matches to define constraints. We therefore need a modified version of the MARVIN algorithm.

Learning constraints

We use a variation of the MARVIN algorithm to learn as many constraints as possible for each example by generalising combinations of terms in the problem text and finding the corresponding pattern for the solution text. A heuristic is used to try to limit the number of combinations: a combination is only valid if the terms are all adjacent. For example, in the problem text “I am called Suky”, “am called” is a valid test combination, while “I called” is not. Each combination is then subjected to generalisation.

Unlike a concept in MARVIN, which is represented by (potentially) a single condition, a constraint is always represented by a conjunction of the relevance and satisfaction condition, i.e. a correct problem/solution example that is relevant to this constraint will satisfy both conditions. Both the relevance and satisfaction conditions can refer to either the problem or the solution. Finally, a constraint can represent a very loose concept, which is suitable for pedagogical purposes but of limited value in checking the correctness of the answer. Other constraints will be needed that are more specific versions of these weaker ones. In this discussion, we limit the constraints we are trying to learn to those where:

1. Each constraint is the most specific test required, and;
2. The relevance condition refers to the problem specification only, and the satisfaction condition refers to just the solution.

Both of these restrictions affect the heuristics used to guide the algorithm. By limiting the generated constraints to the most specific only, we are able to be concise in what we want from the teacher: given a particular example, is it likely that the solution fragment shown is both required and correct. The second restriction guides how we generalise the examples. For any constraint, we want the most general relevance condition possible so that the constraint is maximally applicable. Therefore if a trial constraint turns out to be too general, we begin by trying to specialise the satisfaction condition. Only if that fails do we resort to specialising the relevance condition.

We begin by selecting the combination of words in the problem text that we wish to try to generalise. At this stage we don't know which corresponding terms are relevant in the solution, so we begin by finding this out from the teacher. Next, we begin generalising. As with MARVIN, we start by listing the set of all elaborations of the problem and solution. Next, we choose the first elaboration for each of the problem and solution text, and use them to create a trial concept. Since there may be many, we adopt a generality bias and pick the concept with the largest number of members. We then produce a critical example and present it to the teacher. If it is not correct, we refine the trial concept until we either exhaust the possibilities or the concept is consistent, in which case we build a constraint from it. We then move on to the next combination of terms, and try to build another constraint. At all times we first check whether the target constraint already exists before presenting an example to the teacher.

Example

Suppose we are trying to build a tutor for teaching French to English-speaking students. The concepts that we have already encoded are:

```
Pronoun_English(v1)     := (I, you, he, she, we, they)(v1)

Pronoun_French(v1)      := (Je, tu, il, elle, nous, vous, ils,
                            elles)(v1)
```

```
First_Person_English(v1):= (I, my) (v1)

First_Person_French(v1) := (je, ma, mon) (v1)

Translation(v1, v2)     := ((Je, I),(tu, you),(il, they), (elle,
they)(vous, you),(nous, we), (vous, you),(ma, my),(mon, my),(notre,
our),(ils, they)…) (v1, v2)
```

At this stage we have no constraints in the system. We now present the following problem and solution:

```
"I am called Suky"  "Je m'appelle Suky"
```

We begin by taking the first problem term, "I". We ask the teacher which of the solution terms relates to "I", and are told "Je". We now attempt to generalise. We begin by building the set of all possible elaborations for "I" and "Je". In MARVIN's terminology, this set appears as follows:

```
I(X)                    (1)          Je(Y)                    (2)

Pronoun_English(X)      (3)          Pronoun_French(Y)        (4)
First_Person_English(X) (5)          First_Person_French(Y)   (6)
Translation(Y, X)       (7)
```

The most general substitutions are (3) and (4), and so we pick them. Note that (7) is directional, i.e. Translation (Y, X) means that Y is a translation of X. Therefore, this represents all translations of "I", not all possible translations of a term X, so it has only one member, "Y=je". The trial concept is now:

```
Pronoun_English(X)
Pronoun_French(Y)
```

A critical example is now built, by finding an example that is a member of the trial concept, but does not satisfy ANY of the other conditions from the original, fully elaborated set. In other words, it must be a pronoun, but not first person, the French word must not be a translation of the English. We then present it to the teacher, for example:

```
"you"       "Il"
```

This is incorrect: "il" means "he", which is not a valid example of the concept, so the trial concept is too general. We now attempt to correct the generalisation. As mentioned earlier, we first try to make the satisfaction condition (i.e. the conditions

for the solution) more specific. As for MARVIN, we select another elaboration that was lost in the first generalisation. The most general we can choose is First_Person_French, giving:

```
Pronoun_English(X)
Pronoun_French(Y)
First_Person_French(Y)
```

We again construct a critical example. Again the trial is too general, so we add a further condition, Translation(Y,X). However, we now find that we cannot construct a critical example: There is no English pronoun that is not in the first person, for which there exists a French translation which is a pronoun in the first person. We are therefore forced to backtrack and drop the condition First_Person_French(Y). The process continues by trying the next most general condition, i.e. Translation(Y, X). The trial is now:

```
Pronoun_English(X)
Pronoun_French(Y)
Translation(Y,X)
```

A new critical example is made, i.e. one which satisfies the trial, but does not contain “I” or “Je”, for example:

```
“you”       “tu”
```

This is correct. The teacher now helps to build a new constraint from the trial concept, by adding an appropriate message, and the system translates the new rule into pattern matches and tests, for example:

```
(“You are missing a required pronoun.”
(MATCH PROBLEM (?* (^pronoun_english ?p1) ?*)

(AND
         (MATCH SOLUTION (?* (^pronoun_french ?p2) ?*))
         (TEST SOLUTION (^translation (?p2 ?p1))
 )
)
```

Correcting Overspecialisation

A problem with the generalise-and-test method as described is that it only tests that a constraint is not too general. In MARVIN’s case, overspecialisation is unfortunate but

not catastrophic, because we can simply add an alternative concept later, such that satisfying any description of concept C implies that the new example is an instance of the concept. In our case, this is true of the relevance condition. However, if the satisfaction condition is too specific, the constraint will reject valid solutions.

To overcome this problem, each new training example is first tested against the current set of constraints. If a constraint is violated, it must be reviewed and corrected or rejected. From the above example, suppose we wish to allow *any* valid French phrase that represents the problem statement, including the following. This example will violate the previously learned constraint:

```
"I am called Suky"                    "Ma nom est Suky"
```

Suppose that we wish the above to be accepted. The previously constructed constraint will fail for this input, so must be refined. To do this, we first select the relevant part of the problem, and ask the teacher which parts of the solution are relevant, giving:

```
"I"              "Ma"
```

Next, we build the elaboration list for this example:

```
I(X)                    (1)      Ma(Y)                   (2)
Pronoun_English(X)      (3)      First_Person_French(Y)  (4)
First_Person_English(X) (5)
```

We now build a rule as before. We find that there is no correct solution that can be built using (3), so we are forced to backtrack and consider the next most general clause, First_Person_English(X). This finally yields a new trial of:

```
First_Person_English(X)
First_Person_French(Y)
```

A crucial example is now constructed. To ensure that we do not accidentally pick an example that satisfied the original constraint (and hence this one might be similarly flawed), we add the extra restriction that the example must not satisfy any satisfaction conditions of the original constraint that are not conditions of the new one. In this example, the term Y must not be a pronoun and must not be a translation of X. The example created is:

```
          "I"      "Mon"
```

This is accepted, so the constraint is complete. A new constraint is now built that replaces the old:

```
("The sentence is in the first person. Please check that yours is
too."

(MATCH IS (?* (^first_person_e ?p1) ?*)
          (MATCH SS (?* (^first_person_f ?p2) ?*))
)
```

Conclusions

Using a machine learning algorithm to learn domain constraints such as that described might enables teachers to build constraint-based models by example. It would remove the burden of being able to program constraints and provide a consistent means of reviewing the domain model and making refinements.

We have described how the MARVIN algorithm might be adapted to learn constraints and given a very simple example of how it might work. There are still many questions that must be answered, for example:

- How easy is it for the teacher to comprehend what they are trying to achieve? The example given was for a semantic constraint involving a single term. What about multi-term constraints and syntactic constraints? Can a teacher be reasonably expected to understand and be competent at such a task?
- Is the "build-by-example" approach appropriate? Would it be easier to learn how to write constraints and do that instead?
- What happens when the underlying concept information is incomplete? Does the system simply produce a greater number of more specialised constraints or does it fail altogether?
- Is the method of correcting overspecialisation sufficient or could it cause the system to "flip-flop" between two or more constraint definitions, none of which are satisfied for all possible solutions to the problem?

- It might be more efficient to allow the teacher to enter a list of alternative questions and answers, all with the same meaning, for which any pairing is correct. Could the algorithm be modified to deal with multiple examples at the same time?
- Is it better to train the system on a problem-by problem basis to accept a desired set of problems, or to train it concept-by-concept?
- We have used language translation as an example. Does the approach make any sense in other domain types?

These questions need to be answered before the approach can be considered useful. However, it appears possible for at least some domains.

7.3.2 Constraint editor

The constraint representation introduced in Chapter 4 is a simple language that contains only six constructs: the MATCH function for general pattern matching, the TEST function to test an individual variable value, the TEST_SYMBOL function for performing general pattern matching within a single symbol (rather than a clause) and the logical connectives “AND”, “OR” and “NOT”. Each of the three specialised functions has a fixed set of arguments, and thus a fixed syntax. The AND, OR and NOT connectives have the same syntax as their LISP counterparts.

The match pattern argument to the MATCH and TEST_SYMBOL functions (and, to a lesser degree, the TEST function) also has a restricted syntax. A match pattern is a list of match elements, where each may be a literal, a list of literals, a comparison/assignment of one variable to another and a macro call. Similarly, macros have a single fixed syntax, where the “body” of the macro follows the same syntax as a constraint condition.

Because the language is so restricted, it is highly deterministic. It would therefore be feasible to construct a language-sensitive editor to aid the writing of constraints. This could be similar to the interface used in the LISP tutor, in that it could provide a template for a new constraint, which is expanded by the user. As the author proceeds, new templates are added, for example:

```
<CONSTRAINT>
```

expands to

```
<NUM>
<FEEDBACK>
<REL CONDITION>
<SAT CONDITION>
<CLAUSE>
```

If the user types `MATCH` in the `<REL CONDITION>` slot, it is expanded to

```
(MATCH <SOLUTION> <CLAUSE> <PATTERN>)
```

Scaffolding information could also be provided. For example, the names of all macros could be listed such that these can be “pasted” into the constraint at any time, and doing so would result in a template being provided that is specific to the chosen macro. For example, selecting `^attribute-name` in SQL-Tutor would yield:

```
(^name (<??n> <??a> <??t>))
```

Ideally the constraint editor could itself be an ITS, so that it also provided adaptive help when an author was making errors. In any case, it could test authored constraints and macros for syntactic correctness.

8 Conclusions

Constraint-Based Modelling is a promising new method for representing domain and student models in Intelligent Tutoring Systems. Its efficacy has been demonstrated in the implementation, evaluation and, in some cases, deployment of several CBM tutors including SQL-Tutor, CAPIT and KERMIT. However, CBM tutors have lacked some of the features of the state of the art ITS method, Cognitive tutors. In particular, they are unable give the student specific, tailored advice on how to proceed when she has made an error because they lack a problem solver. Also, building CBM tutors (like all ITS) is hard.

We have explored ways of making CBM tutors more powerful and easier to implement. In doing so we have made several contributions to the field of ITS. Our contributions are now summarised.

8.1 New representation

Ohlsson left open the problems of implementing CBM tutors. In particular, he does not specify how to represent the domain knowledge beyond the basic constraint schematic of {relevance condition, satisfaction condition and feedback}. We have developed a representation for the relevance and satisfaction conditions that is purely pattern matching and have shown its effectiveness in encoding domain models for two domains: SQL and English Language. Further, we have discussed how it might be applied to other domain types such as procedural and graphical domains. The language is complete in that all aspects of the domain model should be able to be encoded using it without the need for external calls. For example, tests for set membership (e.g. checking whether a word in English has been spelled correctly) can

be encoded using macros in the same language. Functions such as arithmetic can be similarly encoded by enumerating the inputs and outputs.

The use of a pattern matching language has several advantages. First, the language is quite simple, consisting only of the three functions MATCH, TEST and TEST_SYMBOL, the logical connectives AND, OR and NOT, and the syntax for defining macros (macro name, arguments, expression). This makes learning the language fairly straightforward and simplifies the authoring of constraints. Second, pattern matching is fast. During evaluation, SQL-Tutor had no problems coping with the demands of multiple users (up to fifteen simultaneously), despite running on a relatively modest server (300MHz PC with 64Mb of memory, running Microsoft Windows NT 4.0). Most student answers are evaluated in under a second. Further improvements could be obtained by compiling the constraints into a dedicated structure such as a RETE network (Forgy 1982). Finally, the new representation is designed to be transparent to the system such that it may reason about the constraints in other ways than simply evaluating them.

8.2 Solution generation

We identified that a shortcoming of our existing CBM tutors was the inability to solve problems, which means that feedback, in terms of "what to do next?" is limited to showing part or all of an ideal solution that may not coincide with the student's attempts. At worst, partial feedback is inconsistent with the student's partially correct answer, and leads to abandonment of the problem.

We designed and implemented an algorithm for generating a correct solution using the constraints. For a null state, this equates to a problem solver. For a student's partial (or incorrect) solution, this algorithm generates a correct solution that is very similar to their attempt. In particular, it employs the same problem solving *strategy* as the student, thus coping with variations between the student's chosen strategy and the author's. We demonstrated in a complex domain (SQL) that this algorithm was able to correct all erroneous solutions from an evaluation study.

A possible drawback to using constraints in this manner is that it imposes the onus of completeness and correctness: if the constraint set is not sufficiently complete and

correct, the problem solver may produce erroneous solutions or fail to terminate. We found that for SQL a considerable amount of work was necessary to attain sufficient correctness/completeness to perform problem solving. However, many of the problems were picked up while using the algorithm to simply build a solution from scratch. Once this was achieved, we tested the algorithm on a small set of student logs. Having corrected the problems encountered with this first set, the number of subsequent corrections/additions necessary to the constraint set to deal with subsequent incorrect solutions was very much smaller. Further, having attained the necessary level of completeness/correctness to deal with a subset of the evaluation students, very few changes were needed to cope with the rest of the students, or indeed a different evaluation population. This suggests that it is feasible that after observing the system for some time and making necessary corrections, the constraints would be sufficient to render the probability of failure negligible.

Finally, while the task of improving the constraint set may seem onerous, a positive side effect is that a more complete constraint set catches more problems and so the tutoring performance of the system might be expected to increase. The reason there were so many additional constraints needed to perform problem solving was chiefly that exhaustively testing the constraint set is a prohibitively large job and had thus never been performed. From our evaluation it appears that in attaining a level of constraint completeness that allowed problem solving, we found and eradicated a large proportion of the omissions in the constraint set. This was demonstrated in the reduction in the number of problems misdiagnosed, from 4.6% to less than 1%. Implementing the problem solver has therefore provided us with a valuable method of testing the completeness of the constraint set.

8.3 Problem generation

A problem affecting all ITS with static problem sets is that they can run out of exercises to present to the student. In CBM tutors the problem is ensuring that the entire curriculum (i.e. all of the constraint set) is covered by problems. Further, problems need to be set over a range of difficulties for all possible combinations of constraints, such that a suitable problem can always be found that fits the student

model. In a domain with a large number of constraints such as SQL this is a huge undertaking.

We have overcome this problem by implementing an algorithm that automatically builds problems from the constraint set. This is an extension of the problem-solving algorithm. Starting with a partial solution that is relevant to a particular constraint (or set of constraints), it applies the problem-solving algorithm and generates a novel SQL statement. An author then produces a natural language problem statement for this new "ideal solution", and the problem is now suitable for presentation to the student. We showed how this algorithm was used to generate 200 problems in the SQL domain in around three hours, a much shorter time than the many days that it took to manually author the 82 problems previously in SQL-Tutor.

There are many different ways to select the next problem to present based on the student model. The method originally used in SQL-Tutor was to select a problem for which the most-often violated constraint was relevant. However, constraints are extremely specific, and there was a high likelihood that no problem would be suitable. We proposed a method for automatically inducing a more high-level student model by identifying groups of constraints of similar meaning, which were either violated or not yet learned. This increases the single violated constraint to a pool of similar constraints, and thus increases the likelihood that a suitable problem can be found.

An alternative strategy is to assess the difficulty of each problem according to how it fits the student model *as a whole*. We developed an algorithm for doing this, which calculates the overall relative difficulty of each problem as the sum of structural (how many concepts are required) and conceptual (what is the student's understanding of each of these concepts) difficulties. We evaluated a version of SQL-Tutor where we used the generated problem set together with this new method of problem selection, and determined that—based on the rate at which constraint errors are reduced—students learn *faster* using this system. However, we did not determine whether the improvement was due to the problem selection method or because there were more problems to choose from.

8.4 Authoring

Building ITS is hard. Previous CBM tutors built by our group were created from scratch. In SQL-Tutor the constraints were implemented in LISP and supported by a substantial body of domain-dependent functions. The tutor engine and domain model were heavily intertwined. CAPIT and KERMIT were written in Visual Basic. KERMIT similarly used custom functions to parse solutions and evaluate constraints, while CAPIT used a generic pattern matcher. However, the problems and solutions in CAPIT are of a very simple structure.

The new constraint representation makes the division between the tutor and the domain knowledge more clearly defined, and arguably reduces the complexity of code (in the pattern matching language) that must be written to specify the constraints and their supporting functions (macros). We took advantage of this to turn SQL-Tutor into an authoring tool, WETAS, for CBM tutors in text-based domains. We generalised the code of the web version of SQL-Tutor by separating out the other domain-dependent parts and making the interface functions data-driven. We demonstrated the flexibility of WETAS by implementing two very different domains: SQL-Tutor and LBITS (in the domain of the English Language). We found that WETAS was suitable for implementing SQL-Tutor and enabled us to rapidly deploy the new LBITS ITS for English. We evaluated LBITS on an elementary school class, who found it easy to use and effective. In future, we would like to include the problem and solution generation algorithms in WETAS.

Finally, we have made initial investigations into induction of Constraint-Based Models using a machine learning algorithm based on MARVIN. While this idea is at a very early stage, it does show some promise and may develop into a useful authoring tool.

8.5 Concluding remarks

The ITS field is maturing, and some methods have achieved a high level of success, such as Cognitive Tutors. These have been shown to be effective for a large number of domains, and have a high level of cognitive fidelity. However, they are very difficult to build and may not be suitable for some domains such as open-ended tasks.

Constraint-based modelling is an alternative method that, like Cognitive Tutors, is also built upon a plausible cognitive foundation. CBM is arguably easier to develop and appears more suitable to open-ended domains. However, current attempts have been limited by their inability to solve problems. Regardless of the modelling method used, building tutors is a large task.

Our aim in this research has been to reduce the effort required to build intelligent tutors without sacrificing effectiveness. We believed CBM was a viable complement to Cognitive tutors, however it had shortcomings that needed to be addressed. We proposed the following four hypotheses:

- **Hypothesis 1**: It is possible to build a constraint-based domain model that contains sufficient information to solve problems and correct student solutions, by adopting a constraint representation that makes all of the logic in each constraint transparent to the system;

- **Hypothesis 2**: Using the representation defined in hypothesis 1, it is possible to develop an algorithm for solving problems and correcting student answers, which does not need further domain information to achieve this;

- **Hypothesis 3**: CBM can also be used to generate new problems that fit the student's current beliefs, and this is superior to selecting one from a pre-defined list;

- **Hypothesis 4**: Because the new representation is domain-independent, it may form the basis of an ITS authoring tool that supports the development of new CBM tutors.

To a student the only major difference between current Cognitive and constraint-based tutors is that the former can solve problems (and thus show the student what the next step is) while the latter cannot. Hypotheses 1 and 2 aimed to show that CBM can indeed be used for problem solving. We produced a representation and solution generation algorithm that worked satisfactorily for two domains. We therefore showed that hypotheses 1 and 2 are true for at least some domains. In doing so, we

have raised the external functionality of a constraint-based tutor to be equal to that of Cognitive tutors.

Our other aim was to make tutors easier to build. Hypotheses 3 and 4 identify two means of doing so: by facilitating the authoring of new problems, and by automating as much of the tutor-building process as possible. We demonstrated that it is possible to use the new solution generation algorithm to build novel structures in the domain being taught, such as novel queries in the case of SQL. The WETAS authoring system automates most of the other functions, the major exception being authoring the domain model. The new representation simplifies this latter task, and we are considering other tools for this purpose too, such as a constraint editor and constraint induction. We believe we have achieved our aim of helping make CBM tutors easier to build, making them a viable alternative to Cognitive tutors.

Intelligent tutoring systems have come a long way since the 1970s. They are now being used in real classroom settings and are producing significant gains in student performance. The next step is widespread deployment, but it has been held back by the huge effort required to build effective systems. We have addressed this by enhancing constraint-based modelling, a simple but effective method, so that it may provide all the domain and student modelling requirements of an ITS. We have developed algorithms and tools that make CBM tutors much easier to build, making CBM a practical tool for ITS deployment. With the number of students ever increasing and the internet opening up the prospective audience of education software, ITS is poised to have an enormous positive impact on education in the near future.

Appendix A. SQL-Tutor evaluation tests

Pretest

Username

Please note down this username. You will be able to access SQL-Tutor only by identifying yourself by this username.

Please answer the following questions, based on the MOVIES database:

1. We want to retrieve titles of all comedies and dramas. Is the following SQL statement correct?

select TITLE from MOVIE where TYPE = 'comedy' or 'drama';	Yes No

2. Show how many dramas were made in each of the following years: 1981, 1982 and 1983. Which of the following statements will achieve that?

Query	Yes/No
select COUNT(*) from MOVIE where YEAR in (1981, 1982, 1983) and TYPE=’drama’	
select COUNT(*) from MOVIE where TYPE='drama' group by YEAR having YEAR=1983 or YEAR=1982 or YEAR=1981;	
select COUNT(*) from MOVIE where YEAR>=1981 and YEAR<=1983;	

3. What is the type of movie that had the highest number of movies made in 1980? Select ALL correct answers.

Query	Yes/No
select TYPE from MOVIE where YEAR=1980 group by TYPE having MAX(COUNT(*));	
select TYPE from MOVIE where YEAR=1980 group by TYPE having COUNT(*) >= all (select COUNT(*) from MOVIE where YEAR=1980 group by TYPE);	
select TYPE from MOVIE where YEAR=1980 and COUNT(*) = (select MAX(COUNT(*)) from MOVIE where YEAR=1980) group by TYPE;	
select TYPE, MAX(COUNT(*)) from MOVIE where YEAR=1980 group by TYPE;	
select TYPE from MOVIE where YEAR=1980 and MNUMBER=MAX(COUNT(*));	

Post-test

Please circle one option:

1. I have not used SQL-Tutor
2. I have used this username while working with SQL-Tutor:

3. I have used SQL-Tutor, but I do not remember my username.

Please answer the following questions, based on the MOVIES database:

4. We need to find the titles of all movies other than comedies. Will the following SQL statement achieve that?

SELECT TITLE FROM MOVIE WHERE TYPE = NOT('comedy')	Yes No

5. We need to find the total number of awards won by comedies in 1983. Which of the following statements will achieve that?

Query	Yes/No
select SUM(AAWON) from MOVIE group by TYPE having TYPE IN ('comedy') and YEAR=1983;	
select SUM(AAWON) from MOVIE where TYPE='comedy' and YEAR=1983;	
select SUM(AAWON) from MOVIE where TYPE='comedy' and YEAR=1983 group by MNUMBER;	

6. Now, we need to find the title of the movie that has won the most awards. Select ALL correct answers.

Query	Yes/No
select TITLE from MOVIE where AAWON = MAX(AAWON);	
select TITLE from MOVIE group by MNUMBER having AAWON = MAX(AAWON);	
select TITLE from MOVIE where AAWON = (select MAX(AAWON) from MOVIE);	
select TITLE from MOVIE group by TITLE having AAWON = (select MAX(AAWON) from MOVIE)	
select TITLE from MOVIE where AAWON>=ALL (select AAWON from MOVIE where AAWON IS NOT NULL);	

Appendix B. Language Builder survey questions

Which puzzle(s) did you play?

- ☐ Scrambled Words
- ☐ Last Two Letters

How were the questions?

- ☐ Too easy
- ☐ About right
- ☐ Too hard

How easy was the software to use

- ☐ Easy to use
- ☐ Okay
- ☐ Hard to use

Did you enjoy using Language Builder?

- ☐ Yes, it was fun
- ☐ It was OK
- ☐ No

How much do you think you learned

- [ ] A lot
- [ ] A little bit
- [ ] Nothing

Appendix C. Publications

In the course of this research we produced the following publications:

1. Martin, B. (1999). Constraint-Based Modelling: Representing Student Knowledge. *New Zealand Journal of Computing* 7(2), pp. 30-38.

2. Martin, B. (2000). Learning Constraints by Asking Questions. In Beck, J. (Ed.), *Proceedings of the ITS'2000 workshop on applying Machine Learning to ITS Design/Construction*, Montreal, pp. 25-30.

3. Martin, B. and Mitrovic, A. (2000a). Tailoring Feedback by Correcting Student Answers. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 383-392.

4. Martin, B. and Mitrovic, A. (2000b). Induction of Higher-Order Knowledge in Constraint-Based Models. In Beck, J. (Ed.), *Proceedings of the ITS'2000 workshop on applying Machine Learning to ITS Design/Construction*, Montreal, pp. 31-36.

5. Martin, B. and Mitrovic, A. (2001a). Increasing Help Adaptability in Constraint-Based Modelling. In, *Proceedings of the AIED2001 Workshop on Help Provision and Help Seeking*, San Antonio, Texas.

6. Martin, B. and Mitrovic, A. (2001b). Easing the ITS Bottleneck with Constraint-Based Modelling. *New Zealand Journal of Computing* 8(3), pp. 38-47.

7. Martin, B. and Mitrovic, A. (2002a). Automatic Problem Generation in Constraint-Based Tutors. In, *Proceedings of the Sixth International Conference on Intelligent Tutoring Systems*, Biarritz, Springer, pp. in press.

8. Martin, B. and Mitrovic, A. (2002b). WETAS: A Web-Based Authoring System for Constraint-Based ITS. In De Bra, P. and Brusilovsky, P. L. (Eds.), *Proceedings of the Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, Malaga, Springer, pp. in press.

9. Mitrovic, A. and Martin, B. (2000). Evaluating Effectiveness of Feedback in SQL-Tutor. In Kinshuk, Jesshope, C. and Okamoto, T. (Eds.), *Proceedings of the International Workshop for Advanced Learning Technologies IWALT2000*, Palmerston North, IEEE Computer Society, pp. 143-144.

10. Mitrovic, A., Mayo, M., Suraweera, P. and Martin, B. (2001). Constraint-Based Tutors: A Success Story. In Monostori, L. and Vancza, J. (Eds.), *Proceedings of the Fourteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems*, Budapest, Hungary, Springer, pp. 931-940.

11. Mitrovic, A., Martin, B. and Mayo, M. (2002). Using evaluation to shape ITS design: Results and experiences with SQL-Tutor. *International Journal of User Modelling and User Adapted Interaction* 12, pp. in press.

Regulation 8(c) of the Degree of Doctor of Philosophy section in the 2002 University of Canterbury Calendar states that "*where the published work has more than one author, it shall be accompanied by a statement signed by the candidate identifying the candidate's own contribution.*" My contributions to these papers were (numbers correspond to list numbers above):

1. The manual hierarchy was created as paid research for Dr Antonija Mitrovic. The idea was therefore hers, but I carried out the research. Dr Mitrovic reviewed the publication;

2. This was entirely my own research and publication;

3. This was my own research, but both the research and the publication draw upon earlier work on SQL-Tutor by Dr Mitrovic. She also reviewed the publication;

4. This research was drew upon on that carried for Dr Mitrovic in (1). I carried out the research, with Dr Mitrovic providing some ideas. She also reviewed the publication;

5. This was my own research, but both the research and the publication draw upon earlier work on SQL-Tutor by Dr Mitrovic. She also reviewed the publication;

6. This was my own research, but both the research and the publication draw upon earlier work on SQL-Tutor by Dr Mitrovic. She also reviewed the publication;

7. This was my own research, but both the research and the publication draw upon earlier work on SQL-Tutor by Dr Mitrovic. She also reviewed the publication;

8. This was my own research, but both the research and the publication draw upon earlier work on SQL-Tutor by Dr Mitrovic. She also reviewed the publication. Jane MacKenzie provided material for the Language Builder domain model;

9. I carried out the analysis of evaluation data as paid research for Dr Mitrovic. I also helped write the paper;

10. I reviewed this paper, and provided some input regarding my research;

11. I was a major co-author of this paper, and performed much of the data analysis.

Signed: _____ Date: _____

Appendix D. Example constraints for Section 5.5

```
(650
 "You do not have all the required attributes in the SELECT clause."

 (and
  (match IS FROM (?* (^table-in-db ?t1) ?*))
  (or-p
   (match IS SELECT (?* (^attribute-of (?n1 ?a1 ?t1))))
   (match IS SELECT ((^attribute-of (?n1 ?a1 ?t1)) ?*))
   (and (match IS SELECT (?* ?before (^attribute-of (?n1 ?a1 ?t1)) ?after ?*))
        (not-p (and (test IS ("(" ?before)) (test IS (")" ?after))))
        )
   )
  )

 (or-p (and
        (or-p
         (and (match SS SELECT
                     (?* ?before2 (^attribute-in-from (?n2 ?a2 ?t2)) ?after2 ?*))
              (not-p (and (test IS ("(" ?before2)) (test IS (")" ?after2))))
              )
         (match SS SELECT (?* (^attribute-in-from (?n2 ?a2 ?t2))))
         (match SS SELECT ((^attribute-in-from (?n2 ?a2 ?t2)) ?*))
         )

        (test SS (^same-attributes (?a2 ?t2 ?a1 ?t1)))
        )
        (or-p
         (and (match SS SELECT (?* ?before2 ?n1 ?after2 ?*))
              (not-p (and (test SS ("(" ?before2)) (test SS (")" ?after2))))
              )
         (match SS SELECT (?* ?n1))
         (match SS SELECT (?n1 ?*))
         )
       )
 "SELECT")


(462
 "Check the comparison operator you used in the WHERE clause to compare the value of
the attribute to a number."

 (and (match IS WHERE (?* (^attribute-p (?n ?a ?t)) (^rel-p ?op1) (^numberp ?c) ?*))
      (match SS WHERE (?* (^attribute-p (?n1 ?a1 ?t1)) (^rel-p ?op2) ?c ?*))
      (test SS (^same-attributes (?a1 ?t1 ?a ?t)))
 )

(or-p
    (and (test IS ("<>" ?op1)) (test SS ("!=" ?op2)))
    (and (test IS ("!=" ?op1)) (test SS ("<>" ?op2)))
    (test SS ((?op1) ?op2))
)
"WHERE")
```

```
(6500
 "Are you sure you need all the attributes in the SELECT clause?"

 (and
  (not-p (match IS SELECT ("*")))
    (or-p
   (match SS SELECT (?* (^attr-name (?n1 ?a1 ?t1))))
   (match SS SELECT ((^attr-name (?n1 ?a1 ?t1)) ?*))
   (and (match SS SELECT (?* ?before (^attr-name (?n1 ?a1 ?t1)) ?after ?*))
        (not-p (test IS ("(" ?before)))
        (not-p (test IS (")" ?after)))
        )
   )

  (or-p
   (match IS SELECT (?* (^attribute-in-from (?n ?a ?t))))
   (match IS SELECT ((^attribute-in-from (?n ?a ?t)) ?*))
   (and (match IS SELECT (?* ?before1 (^attribute-in-from (?n ?a ?t)) ?after ?*))
        (not-p (test IS ("(" ?before1)))
        (not-p (test IS (")" ?after1)))
        )
   )


  (test SS (^attribute-in-db (?a1 ?tdummy)))
  )

 (and
  (or-p
   (match IS SELECT (?* (^attribute-in-from (?n2 ?a2 ?t2))))
   (match IS SELECT ((^attribute-in-from (?n2 ?a2 ?t2)) ?*))
   (and (match IS SELECT (?* ?before2 (^attribute-in-from (?n2 ?a2 ?t2)) ?after2 ?*))
        (not-p (test IS ("(" ?before2)))
        (not-p (test IS (")" ?after2)))
        )
   )

  (or-p
   (and
    ; BIM 21/3/2001 - needs to be in FROM for this to be valid
    (test SS (^attribute-in-from (?n1 ?a1 ?t1)))
    (test SS (^same-attributes (?a1 ?t1 ?a2 ?t2)))
    )
    (test SS ((?n2) ?n1))
    )
   )
  "SELECT")


(350
 "There should be a comma between every two expressions in the SELECT clause."

(and
  (match SS SELECT (?*w1 ?name1 ?name2 ?*w2))
  (not-p (test SS ("AS" ?name1)))
  (not-p (test SS ("AS" ?name2)))
  (or-p
    (test SS (^name ?name1))
    (test SS (")" ?name1))
    (and
        (or-p
            (test SS (^aggrp ?name1))
              (test SS (("ABS" "SIN" "SQRT" "COS" "ATAN" "EXP" "LOG") ?name1))
        )
        (not-p (test SS ("(" ?name2)))
    )
  )
  (or-p
    (test SS (^name ?name2))
    (test SS (^aggrp ?name2))
    (test SS (("ABS" "SIN" "SQRT" "COS" "ATAN" "EXP" "LOG") ?name2))
    (test SS ("DISTINCT" ?name2))
```

```
    (and
      (test SS ("(" ?name2))
      (not-p (test SS (^aggrp ?name1)))
      (not-p (test SS (("ABS" "SIN" "SQRT" "COS" "ATAN" "EXP" "LOG") ?name1)))
    )
  )
)

(and
 (match SS SELECT (?*w1 ?name1 "," ?name2 ?*w2))
 (not-p (match SS SELECT (?*w1 ?name1 ?name2 ?*w2)))
)
"SELECT")

(372
 "Check that you have all the necessary string constants in WHERE - you need to
specify more."
 (and (match IS WHERE (?* (^sql-stringp ?n) ?*))
      (match SS WHERE (?* ?what ?*))
 )

 (match SS WHERE (?* ?n ?*))

"WHERE")


(2730
"Check whether you are comparing the attribute to the right kind of argument in WHERE"

(and
 (match SS WHERE
(?* (^attr-name (?n ?a ?t)) (^rel-p ?op) (^attr-name (?what ?a2 ?t2)) ?*))
 (match IS WHERE (?* (^attr-name (?n2 ?a ?t)) (^rel-p ?op2) (^sql-stringp ?what2) ?*))
 (not-p (match SS WHERE (?* ?n ?op ?what2 ?*)))
 (not-p (match IS WHERE
(?* (^attr-name (?n3 ?a ?t)) (^rel-p ?op3) (^attr-name (?n4 ?a4 ?t4)) ?*)))
)

(test SS ((?what2) ?what))

"WHERE")


(347
 "Check that you use logical connectives (AND, OR) between conditions in the WHERE
clause."

 (or-p (match SS WHERE (?*w1 (^name ?n) (^rel-p ?op) (^sql-stringp ?v) ?c ?*w2))
       (match SS WHERE (?*w1 (^name ?n) (^rel-p ?op) (^numberp ?v) ?c ?*w2))
 )

 (or-p
       (test SS (("AND" "OR" ")") ?c))
       (and
          (match SS WHERE (?*w1 ?n ?op ?v (("AND" "OR") ?lc) ?c ?*w2))
          (not-p (match SS WHERE (?*w1 ?n ?op ?v ?c ?*w2)))
       )
 )

"WHERE")


(454
  "You need to specify an attribute to compare the string constant to in WHERE."

 (match SS WHERE (?* ?what (^rel-p ?op) (^sql-stringp ?c) ?*))

 (test SS (^attribute-p (?what ?a ?t)))

"WHERE")
```

```
(20_A
"When you compare the value of an attribute to a constant, they must be of the same
type."

  (and
    (match SS WHERE (?* (^attribute-p (?n ?a ?t)) (^rel-p ?op) ?c ?*))
    (test SS (^sql-stringp ?c))
  )

  (or-p
    (test SS (^type-p (?a "date")))
    (test SS (^type-p (?a "string")))
  )

"WHERE")


(175
  "Check that you are comparing the string constant to the right attribute in the
WHERE condition."

  (and
     (match IS WHERE
       (?* (^attribute-in-from (?bim1 ?a1 ?t1)) (^rel-p ?op1) (^sql-stringp ?c) ?*))
     (match SS WHERE (?* (^attr-name (?bim2 ?a2 ?t2)) (^rel-p ?op2) ?c ?*))
     (test SS (^type-p (?a2 "string")))
   )

  (or-p
      (and
        (test SS (^attribute-in-from (?bim2 ?a2 ?t2)))
        (test SS (^same-attributes (?a2 ?t2 ?a1 ?t1)))
      )
      (test SS ((?bim1) ?bim2))
   )

"WHERE")
```

References

Ainsworth, S. E., Grimshaw, S. and Underwood, J. (1999). Teachers as Designers: Using REDEEM to Create ITSs for the Classroom. *Computers and Education* 33(2/3), pp. 171-188.

Alexe, C. and Gescei, J. (1996). A Learning Environment for the Surgical Intensive Care Unit. In Frasson, C., Gauthier, G. and Lesgold, A. (Eds.), *Proceedings of the Third International Conference on Intelligent Tutoring Systems*, Montreal, pp. 439-447.

Anderson, J. R., Farrell, R. and Sauers, R. (1984). Learning to Program in LISP. *Cognitive Science* 8(2), pp. 87-130.

Anderson, J. R. and Reiser, B. (1985). The LISP Tutor. *Byte* 10(4), pp. 159-175.

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

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

Anderson, J. R. and Lebiere, C. (1998). The atomic components of thought. MahWah, NJ, Lawrence Erlbaum Associates.

Arroyo, I., Beck, J., Beal, C. and Woolf, B. P. (2000). Macroadapting Animalwatch to gender and cognitive differences with respect to hint interactivity and symbolism. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 574-583.

Ayscough, P. B. (1977). CALCHEMistry. *British Journal of Education Technology* 8, pp. 201-3.

Beck, J., Stern, M. and Haugsjaa, E. (1996). Applications of AI in Education. *ACM Crossroads* 3(1), pp. www.acm.org/crossroads/xrds3-1/aied.html.

Beck, J. and Woolf, B. (2000). High-level student modelling with machine learning. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 584-593.

Blessing (1997). A Programming by Demonstration Authoring Tool for Model-Tracing Tutors. *International Journal of Artificial Intelligence in Education* 8, pp. 233-261.

Bloom, B. S. (1984). The 2 Sigma Problem: the Search for Methods of Group Instruction as Effective as one-to-one Tutoring. *Educational Researcher* 13(6), pp. 4-16.

Bonar, J. and Cunningham, R. (1988). Bridge: An intelligent tutor for thinking about programming. In *Artificial Intelligence and Human Learning, Intelligent Computer Aided Instruction*. Self, J. A. (Ed.), London, Chapman and Hall**,** pp. 391-409.

Brusilovsky, P. L. (1992). A Framework for Intelligent Knowledge Sequencing and Task Sequencing. In Frasson, C., Gauthier, G. and McCalla, G. (Eds.), *Proceedings of the Second International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 499-506.

Brusilovsky, P. L. (2000). Adaptive Hypermedia: From Intelligent Tutoring Systems to Web-Based Education. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 1-7.

Burton, R. R. and Brown, J. S. (1978). A tutoring and student modelling paradigm for gaming environments. *SIGCSE Bulletin* 8(1), pp. 236-246.

Burton, R. R. (1982). Diagnosing bugs in a simple procedural skill. In *Intelligent Tutoring Systems*. Sleeman, D. H. and Brown, J. S. (Eds.), London, UK, Academic Press**,** pp. 157-184.

Carbonell, J. R. (1970). AI in CAI: an artificial intelligence approach to computer-assisted learning. *IEEE transactions on man-machine systems* 11, pp. 190-202.

Cendrowska, J. (1988). PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies* 27(4), pp. 349-370.

Chin, D. N. (2001). Empirical Evaluation of User Models and User-Adapted Systems. *User-Modeling and User Adapted Interaction* 11, pp. 181-194.

Clutterbuck, P. M. (1990). The art of teaching spelling: a ready reference and classroom active resource for Australian primary schools. Melbourne, Longman Australia Pty Ltd.

Corbett, A. T. and Anderson, J. R. (1992). Student Modeling and Mastery Learning in a Computer-based Programming Tutor. In Frasson, C., Gauthier, G. and McCalla, G. (Eds.), *Proceedings of the Second International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 413-420.

Corbett, A. T. and Anderson, J. R. (1993). Student modeling in an intelligent programming tutor. In *Cognitive Models and Intelligent Environments for Learning Programming*. Lemut, E., Du Boulay, B. and Dettori, G. (Eds.), Berlin, Springer-Verlag**,** pp. 135-144.

Cypher, A. (1993). Watch What I Do: Programming by Demonstration. Cambridge, MA, MIT Press.

Deek, F. D. and McHugh, J. A. (1998). A Review and Analysis of Tools for Learning Programming. In Ottmann, T. and Tomek, I. (Eds.), *Proceedings of the ED-MEDIA/ED-TELECOM 98*, Freiburg, Germany, AACE, pp. 320-325.

Dillenbourg, P. and Self, J. A. (1992). People Power: a human-computer collaborative learning system. In Frasson, C., Gauthier, G. and McCalla, G. (Eds.), *Proceedings of the Second International Conference on Intelligent Tutoring Systems*, Montreal, Springer-Verlag, pp. 651-660.

Eliot, C. and Woolf, B. P. (1995). An Adaptive Student Centered Curriculum for an Intelligent Training System. *User Modeling and User-Adapted Interaction* 5(1), pp. 67-86.

Forbus (1997). Using Qualitative Physics to Create Articulate Educational Software. *IEEE Expert* 12(3), pp. 32-41.

Forgy, C. L. (1982). Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1), pp. 17-37.

Gilmore, D. and Self, J. A. (1988). The application of machine learning to intelligent tutoring systems. In *Artificial Intelligence and Human Learning:Intelligent Computer-Aided Instruction*. Self, J. A. (Ed.), London, Chapman and Hall**,** pp. 179-196.

Holt, P., Dubs, S., Jones, M. and Greer, J. (1994). The State of Student Modeling. In *Student Modeling: The Key to Individualized Knowledge-Based Instruction*. Greer, J. and McCalla, G. (Eds.), New York, Springer-Verlag**,** pp. 3-39.

Hsieh, P. Y., Halff, H. M. and Redfield, C. L. (1999). Four Easy Pieces: Development Systems for Generative Instruction. *International Journal of Artificial Intelligence in Education* 10, pp. 1-45.

Johnson, W. L., Rickel, J. W. and Lester, J. C. (2000). Animated Pedagogical Agents: face-to-Face Interaction in Interactive Learning Environments. *International Journal of Artificial Intelligence in Education* 11, pp. 47-78.

Koedinger, K. R., Anderson, J. R., Hadley, W. H. and Mark, M. A. (1997). Intelligent Tutoring Goes To School in the Big City. *International Journal of Artificial Intelligence in Education* 8, pp. 30-43.

Kulik, J. A., Kulik, C.-L. C. and Cohen, P. A. (1980). Effectiveness of computer-based college teaching: a meta-analysis of findings. *Rev. Educ. Research* 50, pp. 524-44.

Lajoie, S. P. and Lesgold, A. (1992). Apprenticeship Training in the Workplace: Computer-Coached Practice Environment as a New Form of Apprenticeship. In *Intelligent Instruction by Computer*. Farr, J. and Psotka, J. (Eds.), Washington D.C., Taylor and Francis**,** pp. 15-36.

Lajoie, S. P. (1993). Computer Envorinments as Cognitive Tools for Enhancing Learning. In *Computers as Cognitive Tools*. Lajoie, S. P. and Derry, S. J. (Eds.), Lawrwnce Erlbaum.

Last, R. W. (1979). The role of computer-assisted learning in modern language teaching. *Assoc. for Literary and Linguistic Computing bulletin* 7, pp. 165-171.

Major, N., Ainsworth, S. E. and Wood, D. J. (1997). REDEEM: Eploiting symbiosis between psychology and authoring environments. *International Journal of Artificial Intelligence in Education* 8, pp. 317-340.

Martin, B. (1999). Constraint-Based Modelling: Representing Student Knowledge. *New Zealand Journal of Computing* 7(2), pp. 30-38.

Martin, B. (2000). Learning Constraints by Asking Questions. In Beck, J. (Ed.), *Proceedings of the ITS'2000 workshop on applying Machine Learning to ITS Design/Construction*, Montreal, pp. 25-30.

Martin, B. and Mitrovic, A. (2000a). Tailoring Feedback by Correcting Student Answers. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 383-392.

Martin, B. and Mitrovic, A. (2000b). Induction of Higher-Order Knowledge in Constraint-Based Models. In Beck, J. (Ed.), *Proceedings of the ITS'2000 workshop on applying Machine Learning to ITS Design/Construction*, Montreal, pp. 31-36.

Martin, B. and Mitrovic, A. (2001a). Increasing Help Adaptability in Constraint-Based Modelling. In, *Proceedings of the AIED2001 Workshop on Help Provision and Help Seeking*, San Antonio, Texas.

Martin, B. and Mitrovic, A. (2001b). Easing the ITS Bottleneck with Constraint-Based Modelling. *New Zealand Journal of Computing* 8(3), pp. 38-47.

Martin, B. and Mitrovic, A. (2002a). Automatic Problem Generation in Constraint-Based Tutors. In Cerri, S. A. and Gouarderes, G. (Eds.), *Proceedings of the Sixth International Conference on Intelligent Tutoring Systems*, Biarritz, Springer, pp. 388-398.

Martin, B. and Mitrovic, A. (2002b). WETAS: A Web-Based Authoring System for Constraint-Based ITS. In De Bra, P., Brusilovsky, P. L. and Conejo, R. (Eds.), *Proceedings of the Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, Malaga, Springer, pp. 543-546.

Mayo, M. and Mitrovic, A. (2000). Using a Probabilistic Student Model to Control Problem Difficulty. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 524-533.

Mayo, M. and Mitrovic, A. (2001). Optimising ITS Behaviour with Bayesian Networks and Decision Theory. *International Journal of Artificial Intelligence in Education* 12, pp. 124-153.

McKenzie, J. (1977). Computers in the teaching of undergraduate science. *British Journal of Education Technology* 8, pp. 214-224.

Michalski, R. (1983). A Theory and Methodology of Inductive Learning. *Artificial Intelligence* 20, pp. 111-161.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review* 63, pp. 81-97.

Mitrovic, A. (1996). SINT- a Symbolic Integration Tutor. In Frasson, C., Gauthier, G. and Lesgold, A. (Eds.), *Proceedings of the Third International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 587-595.

Mitrovic, A. (1998). Experiences in Implementing Constraint-Based Modeling in SQL-Tutor. In Goettl, B. P., Halff, H. M., Redfield, C. L. and Shute, V. J. (Eds.), *Proceedings of the Fourth International Conference on Intelligent Tutoring Systems*, San Antonio, Texas, Springer, pp. 414-423.

Mitrovic, A. and Ohlsson, S. (1999). Evaluation of a Constraint-Based Tutor for a Database Language. *International Journal of Artificial Intelligence in Education* 10, pp. 238-256.

Mitrovic, A. and Hausler, K. (2000). Porting SQL-Tutor to the web. In Peylo, C. (Ed.), *Proceedings of the ITS'2000 workshop on adaptive and intelligent web-based education systems*, Montreal, pp. 37-44.

Mitrovic, A. and Martin, B. (2000). Evaluating Effectiveness of Feedback in SQL-Tutor. In Kinshuk, Jesshope, C. and Okamoto, T. (Eds.), *Proceedings of the International Workshop for Advanced Learning Technologies IWALT2000*, Palmerston North, IEEE Computer Society, pp. 143-144.

Mitrovic, A., Martin, B. and Mayo, M. (2002). Using evaluation to shape ITS design: Results and experiences with SQL-Tutor. *International Journal of User Modelling and User Adapted Interaction* 12(2-3), pp. 243-279.

Munro, A., Johnson, M. C., Pizzini, Q. A., Surmon, D. S., Towne, D. M. and Wogulis, J. L. (1997). Authoring Simulation-Centred Tutors with RIDES. *International Journal of Artificial Intelligence in Education* 8, pp. 284-316.

Murray, T. and Woolf, B. (1992). Results of Encoding Knowledge with Tutor Construction Tools. In, *Proceedings of the AAAI-92*, San Jose, CA, pp. 17-23.

Murray, T. (1997). Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education* 8, pp. 222-232.

Murray, T. (1999). Authoring Intelligent Systems: An analysis of the Starte of the Art. *International Journal of Artificial Intelligence in Education* 10, pp. 98-129.

Murray, T., Piemonte, J., Khan, S., Shen, T. and Condit, C. (2000). Evaluating the Need for Intelligence in an Adaptive Hypermedia System. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 373-382.

Newell, A. and Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In *Cognitive skills and their acquisition*. Anderson, J. R. (Ed.), Hillsdale, NJ, Lawrence Erlbaum Associates**,** pp. 1-56.

Nkambou, R., Gauthier, G. and Frasson, C. (1996). CREAM-Tools: an authoring environment for curriculum and course building in an ITS. In Frasson, C., Gauthier, G. and Lesgold, A. (Eds.), *Proceedings of the Third International Conference on Computer Aided Learning and Instructional Science and Engineering*, Montreal, Springer, pp. 420-429.

Ohlsson, S. and Bee, N. (1991). Strategy Variability: A challenge to models of procedural learning. In Birnbaum, L. (Ed.), *Proceedings of the International Conference of the Learning Sciences*, Charlottesville, pp. 351-356.

Ohlsson, S. and Rees, E. (1991). The function of conceptual understanding in the learning of arithmetic procedures. *Cognition and Instruction* 8(2), pp. 103-179.

Ohlsson, S. (1994). Constraint-Based Student Modeling. In *Student Modeling: The Key to Individualized Knowledge-Based Instruction*. Greer, J. and McCalla, G. (Eds.), New York, Springer-Verlag**,** pp. 167-189.

Ohlsson, S. (1996). Learning from Performance Errors. *Psychological Review* 3(2), pp. 241-262.

O'Shea, T. and Self, J. A. (1983). Learning and Teaching with Computers. Brighton, Harvester Press.

Palmer, B. G. and Oldehoeft, A. E. (1975). The design of an instructional system based on problem-generators. *International Journal of Man-Machine Studies* 7, pp. 249-271.

Petrie-Brown, A. M. (1989). Discourse and dialogue: concepts in intelligent tutoring interactions. *International Journal of Artificial Intelligence in Education* 1(2), pp. 21-29.

Ramadhan, H. and Du Boulay, B. (1993). Programming Environments for Novices. In *Cognitive Models and Intelligent Environments for Learning Programming*. Lemut, E., Du Boulay, B. and Dettori, G. (Eds.), Berlin, Springer-Verlag**,** pp. 125-134.

Rickel, J. and Johnson, W. L. (1997). Intelligent Tutoring in Virtual Reality: A Preliminary Report. In *Eighth World Conference on Artificial Intelligence in Education*. Du Boulay, B. and Mizoguchi, R. (Eds.), IOS Press**,** pp. 294-301.

Russell, D., Moran, T. and Jordan, D. (1988). The Instructional Design Environment. In *Intelligent Tutoring Systems: Lessons Learned*. Psotka, J., Massey, L. D. and Mutter, S. A. (Eds.), Hillsdale, NJ, Lawrence Erlbaum**,** pp. 203-228.

Sammut, C. and Banerji, R. B. (1986). Learning Concepts by Asking Questions. In *Machine Learning: An Artificial Intelligence Approach*. Michalski, R., Carbonell, J. and Mitchell, T. (Eds.), San Mateo, CA, Morgan Kaufman, 2**,** pp. 167-192.

Satava, R. (1996). Advanced simulation technologies for surgical education. *Medical Simulation and Training* 1(1), pp. 6-9.

Self, J. A. (1990). Bypassing the Intractable Problem of Student Modeling. In *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*. Frasson, C. and Gauthier, G. (Eds.), Norwood, NJ, Ablex Publishing Corporation**,** pp. 107-123.

Self, J. A. (1994). Formal Approaches to Student Modelling. In *Student Modeling: The Key to Individualized Knowledge-Based Instruction*. Greer, J. and McCalla, G. (Eds.), New York, Springer-Verlag**,** pp. 295-352.

Self, J. A. (1999). The defining characteristics of intelligent tutoring systems: ITSs care, precisely. *International Journal of Artificial Intelligence in Education* 10, pp. 350-364.

Shute, V. J. and Gawlick-Grendell, L. (1993). An experimental approach to teaching and learning probability: Stat Lady. In Brna, P., Ohlsson, S. and Pain, H. (Eds.), *Proceedings of the World conference on Artificial Intelligence in Education*, Edinburgh, AACE, pp. 177-184.

Shute, V. J., Torreano, L. A. and Ross, E. W. (1999). Exploratory Test of an Automated Knowledge Elicitation and Organization tool. *International Journal of Artificial Intelligence in Education* 10, pp. 365-384.

Sison, R. and Shimura, S. (1998). Student Modeling and Machine Learning. *International Journal of Artificial Intelligence in Education* 9, pp. 128-158.

Soller, A., Goodman, B., Linton, F. and Gaimari, R. (1998). Promoting Effective Peer Interaction in an Intelligent Collaborative Learning System. In Goettl, B. P., Halff, H. M., Redfield, C. L. and Shute, V. J. (Eds.), *Proceedings of the Fourth International Conference on Intelligent Tutoring Systems*, San Antonio, Texas, Springer.

Sparks, R., Dooley, S., Meiskey, L. and Blumenthal, R. (1999). The LEAP Authoring Tool: Supporting complex courseware authoring through reuse, rapid prototyping, and interactive visualizations. *International Journal of Artificial Intelligence in Education* 10, pp. 75-97.

Suraweera, P. and Mitrovic, A. (2000). Evaluating an Animated Pedagogical Agent. In Gauthier, G., Frasson, C. and VanLehn, K. (Eds.), *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, Montreal, Springer, pp. 73-82.

Suraweera, P. and Mitrovic, A. (2001). Designing an Intelligent Tutoring System for Database Modelling. In Smith, M. J. and Salvendy, G. (Eds.), *Proceedings of the 9th Int. Conf Human-Computer Interaction International (HCII 2001)*, New Orleans, Lawrence Erlbaum Associates, pp. 745-949.

VanLehn, K. (1983). On the Representation of Procedures in Repair Theory. In *The Development of Mathematical Thinking*. Ginsburg, H. P. (Ed.), New York, Academic Press**,** pp. 201-252.

Verdejo, M. F., Fernandez, I. and Urretavizcaya, M. T. (1993). Methodology and design issues in Capra: an environment for learning program construction. In *Cognitive Models and Intelligent Environments for Learning Programming*. Lemut, E., Du Boulay, B. and Dettori, G. (Eds.), Berlin, Springer-Verlag**,** pp. 156-171.

Weber, G. (1993). Analogies in an intelligent programming environment for learning LISP. In *Cognitive Models and Intelligent Environments for Learning Programming*. Lemut, E., Du Boulay, B. and Dettori, G. (Eds.), Berlin, Springer-Verlag**,** pp. 210-219.

Winograd, T. (1975). Frame representations and the declarative-procedural controversy. In *Representations and understanding*. Bobrow, D. and Collins, A. (Eds.), New York, Academic Press**,** pp. 185-210.

Woolf, B. and Cunningham, P. A. (1987). Multiple Knowledge Sources in Intelligent Teaching Systems. *IEEE Expert* 2(1), pp. 41-54.

Yacef, K. and Alem, L. (1996). Student and Expert modelling for Simulation-based Training: A cost effective framework. In Frasson, C., Gauthier, G. and Lesgold, A. (Eds.), *Proceedings of the Third International Conference for Intelligent Tutoring Systems*, Montreal, Springer, pp. 614-622.