A logic programming e-learning tool for teaching database dependency theory.

Paul Douglas
School of Informatics, University of Westminster

Steve Barker
King's College, London

A Logic Programming E-Learning Tool For Teaching Database Dependency Theory

Paul Douglas

University of Westminster, London, UK

P.Douglas@wmin.ac.uk

Steve Barker

King's College, London, UK

steve@dcs.kcl.ac.uk

Abstract

In this paper, we describe an "intelligent" tool for helping to teach the principles of database design. The software that we present uses PROLOG to implement a teaching tool with which students can explore the concepts of dependency theory, and the normalization process. Students are able to construct their own learning environment and can develop their understanding of the material at a pace that is controlled by the individual student.

1 Introduction

We describe a tool that we have developed and have used to help university-level students to learn certain essential notions in database schema design, specifically the *normalization* [1] process, which is based on the underlying concept of *dependency theory* [1]. We regard the learning tool as a piece of intelligent software, where the term "intelligent" is interpreted by us as the capability of responding to a student's input (in the form of a database design problem *chosen by the student*) with a solution based entirely on that input. There are no "standard" problems or solutions provided with the software, and the software is able to explain each step of the solution process if the student chooses to exploit this option. It follows that the software is capable of providing either a quick check of a student's own work, or a fuller teaching facility.

PROLOG is used for the implementation of the software to provide the capability of checking a student's work and either confirming that the work is correct, or indicating why it is not. PROLOG has been widely used for implementing items of educational software (see, for example, [2] and [3]) and is appropriate for the teaching tool that we have developed because it permits a rule-engine to be exploited to intelligently interpret and respond to student inputs. The database

71

design algorithms that we have chosen to implement also have a natural translation into PROLOG code.

A number of excellent textbooks on dependency theory already exist. Nevertheless, although textbooks can offer very good coverage of the material on dependency theory, they offer only limited forms of interactivity and limited scope for students to test their own understanding of database design principles. Many textbooks provide no practical exercises and, even when they do, these exercises are often limited in size and sophistication. Moreover, textbooks that do provide exercises do not necessarily provide "solutions", so students cannot determine whether they were able to "solve" the problems.

Some courses that teach relational databases take the approach that the use of a schema design tool will almost always deliver a schema that is in *third normal form (3NF)* [1], and that teaching dependency theory is not really necessary (see, for example, [4]). However, we disagree with this point of view. There are many aspects of relational database technology that are directly related to the data dependencies that exist within a database, and students cannot properly consider these issues without a proper understanding of dependency theory. The use of design tools also suggests that a relational database schema has been somehow finalized once it has been put into third normal form, leaving students with even less understanding of normal forms beyond 3NF. Students will not be able to critically evaluate alternative designs, or make informed choices about levels of normalization, if they do not understand the principles upon which design decisions have been based. *Functional dependencies* [1] are also important for a proper understanding of the concepts of *candidate keys*, *superkeys*, constraints, and the theoretical foundations of relational database systems.

The PROLOG programs that we have developed are able to lead a student through the process of decomposing relations to satisfy the requirements of a particular normal form for database design. The tool is able to explain the steps that are being taken to generate a decomposition, and can thus provide a solution to a given decomposition problem whilst also providing an explanation about how it was achieved. A fairly simple interface program written in Java allows the students to enter un-normalized relations and go through the steps of normalizing the relations to a "higher" normal form without having to interact directly with PROLOG themselves. In this way, students are able to compare not only a solution to a problem with their own, but to see a whole method for the problem worked out, step by step, and to call on a textual explanation facility at any point at which they do not understand the processes that have taken place. Because our tool enables students to freely select inputs, it enables them to work on problems of their own devising, at a level of difficulty exactly appropriate to their own level of understanding.

The remainder of this paper is organized as follows. In Section 2, a number of preliminary issues are discussed. In Section 3, the implementation of the teaching tool is described. In Section 4, a sample user session is described. In Section 5, the evaluation of the software is considered. Finally, conclusions and suggestions for further work are given in Section 6.

We assume that the reader has a knowledge of the basic notions and notations that are typically used in discussions on dependency theory; otherwise, we suggest [1] for all necessary background information.

2 Preliminaries

Our approach to developing our learning tool for database design initially involved us adopting a *phenomenographic* method [5] for information gathering on students' understanding of concepts in dependency theory. By conducting 'dialogue' sessions with students we identified the strategies students used to understand the basic concepts. From our review of the notes taken at the dialogue sessions, we were able to develop a prototype system for supporting students in learning about dependency theory.

As the software evolved, we made increasing use of Gagne's *event-based model of instruction* [6] to decide what material a user of the tool should be offered and the order in which information ought to be presented to a learner. Thus, different levels of learning guidance are available to meet the requirements of an individual student, and learning takes place in a student-centred, interactive way.

In overview, our learning tool includes implementations of the following algorithms. We use Ullman's FD-closure algorithm [1]; we use Beeri and Honeyman's algorithm [7] for checking dependency-preservation after decomposition; we use Loizou and Thanisch's approach [8] for checking for a lossless-join decomposition; we use Ullman's method for finding a minimal cover for a set of FDs [1]; we use Gottlob's method [9] for computing a cover for the projection of a set of FDs onto a subschema of the decomposition; and we use Luchessi and Osborn's key finding algorithm [10] to identify candidate keys. For the decomposition algorithms, we used the proposal in Ullman [1] for generating 3NF schemes, and Tsou and Fischer's approach [11] for generating BCNF schemes.

In our approach, an n-ary *relational scheme* of the form

$$R(A_1, \ldots, A_n)$$

where R is the name of the relation and $A_1, \ldots, A_n$ is a set of attributes is represented in our PROLOG implementation by using a list: $[A_1, \ldots, A_n]$.

Moreover, given a functional dependency of the form:

$$A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n$$

where A_i ($i \in \{1, .., m\}$) and B_j ($i \in \{1, .., n\}$) are attributes, we use pairs of lists and define an operator for $\rightarrow$, to wit:

$$[A_1, \ldots, A_m] \rightarrow [B_1, \ldots, B_n]$$

3   Implementation

Our implementation of the database design tool includes a GUI that sits on top of the PROLOG implementation of the database design algorithms.

The interface is intended to be simple to use; it is menu-based and all data that is entered is case-insensitive. Users are prompted throughout a session for the correct data to enter, and can return to the main menu at any time. It is possible to enter multiple schemas, save them, and return to them later within a session. Sessions can also be retained in a file, and can therefore be suspended and resumed.

The interface program is written in Java. Java has many advantages for this kind of application. It is widely used, it has comprehensive Internet support (see below), and it is easy to access applications written in a variety of other languages (through the *Java Native Interface (JNI)* mechanism).

We use XSB-Prolog [12] to implement the main logic programs that implement the database design algorithms. XSB runs on a number of platforms and offers excellent performance that has been demonstrated to be far superior to that of traditional Prolog-based systems [13].

Calls to XSB from the interface program are handled by the YAJXB [14] package. YAJXB makes use of Java's JNI mechanism to invoke methods in the C interface library package supplied by XSB. It also handles all of the data type conversions that are needed when passing data between C and Prolog-based applications. YAJXB effectively provides all the functionality of the C package within a Java environment.

Although we have used YAJXB in our implementation, we note that a number of alternative options exist. Amongst the options that we considered were Interprolog, a Java-based Prolog interpreter (e.g., JavaLog), or a Sockets-based, direct communication approach. Unfortunately, each of these approaches has its own distinct drawbacks when compared with the approach that we adopted. Interprolog does work with XSB, so we could still take advantage of the latter's performance capabilities. However, Interprolog is primarily a Windows-based application. All of our development was done on a Sun Sparc/Solaris system; YAJXB, though primarily configured for Linux, compiles easily on Solaris. JavaLog

was discounted because we felt that it did not offer sufficient flexibility compared with XSB. Finally, using sockets would give us a less portable application because it would involve considerably more application-specific coding. Overall, we felt that the straightforwardness of the YAJXB interface makes it preferable to the Interprolog approach so far as interfacing with XSB is concerned. Moreover, XSB's highly developed status and excellent performance make it more desirable in this context than a Java/Prolog hybrid.

The C library allows the full functionality of XSB to be used. A variety of methods for passing Prolog-style goal clauses to XSB exists. However, we generally found that the string method worked well. This method involves constructing a string σ in a Java String type variable, and using the $xsb\_command\_string$ function (or similar) to pass σ to XSB. This approach allows any string that could be entered as a command when using XSB interactively to be passed to XSB by the interface program. YAJXB creates an interface object; the precise method of doing this is a call like:

$$i = core.xsb\_command\_string \ (command.toString());$$

where the assignment, as one would expect, handles the returned error code. Variations on this method allow for the return of data where relevant.

4   A Learning Session

Users invoke the software by using the Java JRE. On invoking the software, the system will respond with the opening menu:

<div align="center">

MAIN MENU

</div>

1. Enter a Schema

2. Help

3. Exit

<div align="center">

Enter Choice (1-3):

</div>

The "help" option gives some general guidance on how to use the system; the "exit" option terminates the program. Having invoked the system, the user will normally enter a schema. The system will first prompt the user to enter the names of the attributes:

Enter attribute names, using spaces to separate them.
Names must be single characters or strings:

If the required schema attributes are (a, b, c, d) then the user will respond, to the request for input, with something like:

a b c d

The next step is for the functional dependencies to be entered. The user is first prompted to enter a determinant, then its dependent attributes. The process will be repeated for each determinant. The system loops around these input processes until a blank line is entered: for each determinant the user is repeatedly prompted to enter another dependent attribute until a blank line is entered; the user is then invited to enter another determinant, and this process in turn repeats until a blank line is entered. For example, for the functional dependency $a \rightarrow bc$ we have:

Enter a determinant
If multivalued, use spaces as separators: a

Enter a dependent attribute
If multivalued, use spaces as separators: b

Enter a dependent attribute
If multivalued, use spaces as separators: c

Enter another dependent attribute
(return to end):

. . .

When the process of describing functional dependencies is complete, the system will respond with a display of the information entered and another menu. All functional dependencies are displayed in *right reduced form* i.e., with a single set of attributes on the right-hand side of an FD. For example:

Your schema has attributes: [a,b,c,d]

and FDs: [a]→[b], [a]→[c], [c]→[d]

CHOOSE AN OPTION:

1. 3NF decomposition

2. BCNF decomposition

3. Help

4. Exit

<div align="center">Enter Choice (1-4):</div>

The "exit" option returns the user to the previous menu; the "help" option gives some general information about the decomposition process. The example that follows gives some sample output if 3NF decomposition is selected from the menu:

```
Finding a minimal cover
At each step, enter ? for help or CR to continue...
...checking right reduction
...checking left irreducibility
...checking redundant FDs
Decomposing...
...checking lossless join property

The following 3NF subschemata
give the dependency preserving
decomposition of your schema:

t_1
    [a,b,c]
    one key: [a]
t_2
    [c,d]
    one key: [c]
```

Having generated the 3NF decomposition, the user can then return to the main menu, and either enter another schema, or exit the system.

5 Evaluating The Software

Our database design teaching and learning tool has been formatively evaluated. For the formative evaluation, we sought comments from several colleagues involved in teaching database management at the University of Westminster; these

were our "expert reviewers" [15]). We additionally worked with a small group of post-graduate students who were learning about dependency theory at the time at which they used the software; these students tested the software during tutorials over two consecutive weeks, and in several additional sessions. A number of suggestions made by the expert reviewers and the volunteer students were used to make minor modifications to our initial design e.g., modifications of the interface.

We then conducted a program of small-group testing with some final year undergraduate students who were also studying dependency theory as part of a database design module. We gathered feedback about the software by using observations and informal "interviews". This involved one of the authors sitting with the students and asking them to articulate their feelings about the software, and asking the students to complete a questionnaire at the end of the trial period, which asked them how, in general, they had found the software to use, and how they felt it compared with the traditional textbook alternative.

The students all reported that the software was useful in terms of helping to develop their understanding of dependency theory, and all agreed that the facility for testing their own solutions to normalization problems was motivating to use and important in developing understanding. They were unanimous in concluding that the tool was a major improvement, in terms of carrying out practical exercises, over the textbook.

It is not perhaps surprising that the overall feedback was so positive. Using something new is always more interesting and students like to use computers. Because of our desire to get the feedback, the students probably found the tutorials relating to the dependency theory material a more positive experience than those provided to support the rest of the module (the use of a couple of volunteer helpers from the earlier post-graduate test group resulted in a much higher staff-student ratio than usual).

6   Conclusions and Further Work

We have developed a teaching and learning tool for helping university students to learn some aspects of dependency theory. The results of discussions with the students who have used the software suggest that the tool is of value to students learning about dependency theory. However, much more extensive testing of the software will be necessary (e.g., a summative evaluation) before any firm conclusions can be drawn about its educational value.

There are several ways in which the tool could be further developed. In particular, we plan to produce a web-based interface, which will make the tool both easier to use, and more widely accessible. It would additionally enable us to

improve the availability of the explanation facility, which could be read in pop-up help windows at all stages of the normalization process. We also intend to develop our tool to assist students in their learning of multivalued and join dependencies and the normal forms that are associated with these types of dependencies.

7 References

[1] J. Ullman, Principles of Database and Knowledge-base Systems, Computer Science Press, 1989.

[2] Yazdani, M., New Horizons in Educational Computing, Chichester: Ellis Horwood, 1983.

[3] Nichol, J., Briggs, J., and Dean, J., Prolog, Children and Students, London: Kogan-Page, 1988.

[4] B. Byrne, Top Down Approaches to Database Design Tend to Produce Fully Normalised Designs Anyway, Proceedings of TLAD, 2003.

[5] F. Marton and P. Ramsden, What does it take to improve learning?, Improving Learning: New Perspectives, Kogan Page, 1988.

[6] R. M. Gagne, The Conditions of Learning, Holt, Reinhart and Winston, 1970.

[7] C. Beeri and P. Honeyman, Preserving Functional Dependencies, SIAM Journal of Computing, 10(3), 1981.

[8] G. Loizou and P. Thanisch, Testing a Dependency-preserving Decomposition for Losslessness, Information Systems, 8(1), 1983.

[9] G. Gottlob, Computing Covers for Embedded Functional Dependencies, PODS, 1987.

[10] C. Lucchesi and S. Osborn, Candidate Keys for Relations, Journal of Computer and System Sciences, 17(2), 1978.

[11] D. Tsou and P. Fischer, Decomposition of a Relation Scheme into Boyce-Codd Normal Form, SIGACT News 14(3), 1982.

[12] K Sagonas, T. Swift, D. Warren, J. Freire and P. Rao., The XSB System Version 2.0, Programmer's Manual, 1999.

[13] K Sagonas, T. Swift, and D. Warren., XSB as an Efficient Deductive Database Engine, ACM SIGMOD Proceedings, 1994.

[14] S. Decker, Yet Another Java XSB Bridge, http://www-db.stanford.edu/%7Estefan/rdf/yajxb/

[15] M. Tessmer, Planning and Conducting Formative Evaluations, Kogan-Page, 1993.