Design and Implementation of a Database Design Aid Using VP-Expert

Il-Yeol Song, Samuel D. Strum, and Carl Medsker

College of Information Studies, Drexel University, Philadelphia, PA 19104

Abstract

We discuss the design and implementation of a database modeling aid called Database Designer (DBD) using VP-Expert. DBD is an expert system to help novice users design a proper database by asking questions of the user and thus incrementally building a structure for an extended ER diagram. We discuss our approach to system design for DBD and implementation issues specific to VP-Expert such as combining backward chaining and forward chaining, procedural techniques, and transparency and visibility of VP-Expert statements. Our study indicates that, by applying a systematic, structured approach, properly commenting the code, and logically grouping rules, an ES programming approach is viable in designing a complex procedural design application like DBD.

1 Introduction

The purpose of this paper is to present the Database Designer (DBD), a tool for aiding novices in the design of database applications. Since proper database design includes many constraints and rules, the tool utilizes an expert system (ES) approach. In general, a database designer questions a user about the enterprise under consideration, analyzes the user's responses, creates a conceptual model, and translates that model into a working application. DBD reflects this sequence.

Powerful and sophisticated database systems are readily available to novices. However, the spread of database design theory and conceptual modeling techniques has not kept pace. Software documentation rarely addresses design theory and thus applications vary in quality and integrity. Furthermore most CASE tools support database professionals rather than novices.

DBD is one of a suite of tools under development that extends the EER Modeling Aids (EERMA) for novice database designers using structured semantic design approach [1]. Based upon the Entity Relationship (ER) model [2] and Extended Entity Relationship (EER) model [8], EERMA uses structured English template sentences to translate business rules into EER models.

Research and experience have shown that the most intriguing, vague, and difficult part of the database design process is translating the semantics of an enterprise into a conceptual model [4, 7]. DBD, although partially completed, is a successful aid in this process.

The paper is organized as follows: we present our system design in Section 2; system architecture in Section 3; implementation problems and techniques in Section 4; related applications in Section 5, and conclusion in Section 6.

2 System Design

DBD was implemented in VP-Expert and dBase 111+ because both are popular, reasonably priced and readily available.
NOVICE USER
HAS A CONCEPTUAL IDEAS OF ENTERPRISE TO BE MODELED IN DATABASE.

Input

DATABASE DESIGNER
(KNOWLEDGE ENGINEER)

Output

ASKS QUESTIONS TO USER REGARDING ENTITIES AND RELATIONSHIPS BETWEEN ENTITIES OF ENTERPRISE.

TRANSLATES RESPONSES AND CREATES LOGICAL MODEL OF DATABASE INCLUDING: VALID ENTITIES; RELATIONSHIP DEGREE; CARDINALITY, AND PARTICIPATION; ENTITY IDENTIFIERS AND ATTRIBUTES; ABSTRACTION, AND EXISTENCE DEPENDENCE.

A DATABASE OR REPORT WHICH CAN EASILY BE USED TO CONSTRUCT A VALID, COMPLETE EER DIAGRAM.

FIGURE 1 INPUT/OUTPUT DIAGRAM

2.1 Conceptual Framework

The purpose of DBD, as depicted in Figure 1, is to mimic an actual database professional in an interaction with a user. The database professional helps users create an EER model, the output, from semantic descriptions, the input.

2.2 Object Model

Figure 2 shows the structure of DBD objects. The system consists of four database constructs: entity, relationship, abstraction, and existence dependency. Instances of these objects will be the actual entities, relationships, etc. of the user's enterprise. DBD allows an unlimited number of attributes for any given entity and any number of entities per relationship.

2.3 System Flowchart

The flowchart in Figure 3 represents DBD processes. The part shown works as follows:

1) The first phase is Entity Initialization. The user is asked for information about relevant entities and attributes. For a particular entity, attributes are requested until the user indicates completion. Then an entity identifier is requested. Options are given for this identifier: pick one existing attribute, create a new identifier (e.g. House_ID), or pick two or more existing attributes. Then another entity is requested and processed and the cycle continues until the user indicates no more entities exist.

2) The responses from Step 1 are passed to the Entity Verification phase, which ensures that the specified entities and attributes adhere to accepted EER modeling rules (entities do not have same name, entities have at least one attribute, etc.)

In the complete process, each entity is presented to the user in juxtaposition to every other entity and the user describes the relevant relationships. Finally, the relationship inputs are verified.
FIGURE 2 OBJECT MODEL

FIGURE 3 DECOMPOSED I/O DIAGRAM
2.4 Expert System Design

DBD is not a "standard" backward chaining ES. Rather, it is a design/configuration application, relying heavily on forward chaining and procedural methods [3]. DBD uses forward chaining inference, since the explicit solution is unknown, although it may be any valid EER Diagram, and the initial data, (entities, attributes, and relationships) drive the system.

DBD is also procedural because in the process of questioning users about an enterprise the sequence of questions and actions is critical in acquiring the most correct responses. For example, DBD requests attributes of entities immediately after the user specifies an entity while the concept is fresh in mind. Also, when verifying input, certain items should be checked first to avoid complications later. For example, if a user has entered two entities with the same name, the system checks this first. If the user chooses to delete the repeated item, the associated attributes are never specified. DBD is, therefore, a data driven, procedural-like design application.

2.5 Output Design

The output of the system is database files. Initially, these files directly reflected the object structure shown in Figure 2. The entity file, for instance, contained an entity name field and a limited number of attribute fields, (i.e. att1, att2, ... att10). However, a database of this nature limits the number of attributes and may contain many null values. As a remedy, two database files for entities were created: one stores the entity name and identifier, and the other relates an entity with each of its attributes as shown in Figure 4. Relationships, Abstractions, and Existence Dependencies have a similar structure.

```
ENTITY_FILE (ENTITY, ID)
ATTR_LIST (ENTITY, ATTRIBUTE)
REL_ATTR (REL_NAME, ATTRIBUTE)
REL_DEGREE (REL_NAME, DEGREE)
RELATIONSHIP (REL_NAME, ENTITY, PARTICIPATION, CARDINALITY, ROLE)
```

FIGURE 4

DATABASE STRUCTURES IN DBD

3 System Architecture

DBD uses a four step process to elicit enterprise specifications from a user, verify the input items, request relationships between items, and verify relationships. A detailed DBD process, consisting of four modules, each with its own input and output, is shown in Figure 3. Notice that each module further verifies the enterprise, resulting in a legal and complete model. Ideally, a system such as this would be implemented in one complete programming environment. VP-Expert, however, does not suffice as that complete environment. Since the entities, attributes, relationships, etc. are represented in database files, a dBase III+ program was used to create the initial entity and relationship databases. The VP-Expert program transfers data to and from dBase files during verification. As a result, a user of DBD must exit dBase after entering initial items and enter VP-Expert to run the verification program. An embedded system would compile the tasks into one shell and eliminate needless switching.

4 VP-Expert Discussion

In this section, we discuss the various aspects of VP-Expert as an implementation tool for DBD. VP-Expert allows one dimensional arrays and forward chaining, which were both necessary in this project. Still, it seemed VP-Expert would have to be manipulated and twisted in uncommon ways to handle the application, and that was indeed the case.

4.1 Forward Chaining
An ES for design applications should utilize forward chaining, since its desired goals are not known in advance or are too numerous to list. Forward chaining is most useful in creating data driven processes and ESs with unknown goals. In VP-Expert, the WHENEVER statement is used for forward chaining. It works as follows:

When a variable in the condition of a WHENEVER rule changes value, the WHENEVER rule fires immediately, then control returns to where it left off. Forward chaining rules provide procedural control.

Backward chaining rules, whose goals are searched at the proper time, enable and disable certain variables. With forward chaining, these results cause or do not cause WHENEVER rules to fire.

Harmon and Sawyer [3] suggest these steps for creating a forward chaining ES:

1) Define the problem
2) Write Code to obtain the facts
3) Write a first set of rules
4) Give the system a way to stop
5) Control rule execution
6) Further development

Control of rule execution is of the utmost importance in this system. Initially, a template routine was constructed showing basic control execution from which a general system was created. DBD was developed with these constraints imposed by VP-Expert [9]:

1) For a forward chaining WHENEVER rule to fire, the variable in the condition part must be modified.
2) For a backward chaining rule to fire, the goal variable must be UNKNOWN.
3) Any valid statement is legal in the THEN part of a backward chaining rule, but the first statement must assign a value to the goal variable.
4) On the contrary, in a WHENEVER rule, there is no need to assign a value to a goal variable in the THEN part, plus, the FIND statement is not allowed in the THEN part of the rule.

4.2 VP-Expert Strategy

Acknowledging the above constraints, the project strategy applied forward chaining whenever possible. However, a FIND statement was necessary for consulting the user and this required, according to constraint 4 above, backward chaining.

The strategy is as follows: (See Appendix for the code.) Specific variables are set during program initialization. These variables are RESET if certain conflicts occur. When a FIND statement with these variables is met, the corresponding ASK statement is instantiated and the user is questioned.

Often, two items in the system must be compared, (e.g. entities and attributes with the same names.) This comparison is handled with a nested loop as follows:

A loop is initiated which iteratively assigns the first variable to be compared. In that loop a FIND calls a rule, (a loop rule), which always fires and is superfluous with regard to variable assignment. In the THEN part of this "loop" rule, a second loop is initiated which iteratively assigns the second variable to be compared. A "comparing" rule is fired which checks if the two variables are equal. If that "comparing" rule fires, (the variables are equal), the goal variable is set. Control then returns to the next step in the "loop" rule, where another FIND is encountered. The corresponding rule that FIND triggers checks the value of the previous goal variable to see if the two variables are equal. If so, the goal variable of this rule is set indicating a conflict has occurred. The user is
informed of the problem, and a response is requested. A forward-chaining WHENEVER rule is immediately fired when the user responds, and certain actions may be taken on the data. Finally, subsequent variables are RESET and even more rules may be triggered, depending on the user's response. This series of actions is the basic approach taken in each subroutine in both the entity and relationship verification programs.

4.3 Transparency and Visibility

Given the procedural aspects of this application, what are the advantages, if any, of VP-Expert? Inherent differences exist between conventional and declarative programming systems [5]. This project illustrates Pedersen's statement that knowledge bases are transparent because rule meaning is relatively separate from location in the knowledge base. The rules in the Appendix are each an independent module of knowledge, providing far more transparency than conventional programming. When debugging conventional systems, a programmer must constantly recall previous actions or program states, which can be tedious and complex. The problem is less burdensome in ES programming if the conditions of each rule explicitly and completely state the requirements for a rule to fire.

However, the lack of visibility that Pedersen mentions [6] in ES shells, especially VP-Expert, is a problem when creating and/or debugging a program. Consider a simple VP-Expert program. In the ACTION section at the beginning of the program, a MENU statement may be needed, which is attached to an ASK statement at the end of the program. In the ACTION part, a FIND statement starts a search. If the goal variable is not already defined in memory, the system searches the rules. If not satisfied, the search continues to an ASK statement, again at the end of the program. In DBD, the situation is more complex. For example, as shown in the Appendix, a FIND statement within a FOR loop in the ACTION section may trigger a rule, which actually initiates another loop. Within that loop, another FIND statement may be encountered. That FIND statement may check a condition, set the goal variable, and then proceed and trigger another FIND statement. The search of this FIND may ASK the user a question. The ASK, of course, is typically at the bottom of the program listing and the corresponding MENU statement is at the very top. The answer to that ASK may trigger a WHENEVER rule. Finally, control is returned to the second FIND statement. Several practices ease this lack of visibility in ES programming:

1) Proper commenting of rules is essential.
2) A system of grouping rules for readability and maintenance, not necessarily efficiency, is mandatory.
3) Variables must be aptly and sensibly named.
4) A systematic, structured approach enhances program growth.

4.4 VP-Expert Problems

This section discusses problems encountered in this project. First, VP-Expert cannot delete a database record. To compensate, a blank value in a particular field could be used to mark a record for deletion. However, VP-Expert also will not write blank values to a database record. So the word "NONE" was used to mark a record for deletion by a dBase "housekeeping" program.

VP-Expert exhibited another peculiarity. Even when a goal variable was UNKNOWN, a FIND statement would not always search for the goal rule, as if the variable was already defined in working memory. Instead, it seemed goal variables needed to be RESET immediately prior to the corresponding FIND (note Appendix). Debugging and resolving this quirk was difficult.
Other problems occurred using VP-Expert's "?" function. A user input of "?" indicates UNKNOWN which, in this application, created a complex situation. The response to this question had to be acted upon with a WHENEVER rule. However, in order for the question to be asked, it must have been UNKNOWN beforehand. If the user enters "?", the variable remains UNKNOWN and since it does not change, the WHENEVER rule will not fire. This situation was resolved by changing the ordering in which items were checked.

A more general problem was rule grouping. The optimal goal was to group backward chaining rules together, all rules in the same process together, and all rules of the same level together. A complete accommodation was impossible, so the final grouping system was:

ACTION -> LR -> GR -> BR -> FR -> ASK

where
LR = Looping Rule
GR = General Rule
BR = Backward Chaining Rule
FR = Forward Chaining Rule
ASK = ASK Statements

Finally, whenever someone creates a database model, the output must be saved in a file, but with what name? DBD asks the user for the overall concept of the enterprise and stores the response in the first record of the entity file, (entity_f.dbf.) The verification routine uses the general file names (entity_f.dbf and attlist.dbf.) After completing verification, a dBase routine creates file names using the first record of the entity file. The verified enterprise data is copied to these new files and then a blank structure (entity_f.tmp and attlist.tmp) is copied over the general "work" files (entity_f.dbf and attlist.dbf.). This dBase routine is under development.

5 Related Applications

Novices need data modeling aids to design databases because the growth in database professionals has not kept pace with the spread of microcomputer based database software. Databases with minimal redundancy and appropriate associations of tables which yield correct answers to queries are essential for correct decision making. DBD is one of a set of EERMA based tools intended to address this need. Other tools, in varying stages of development, include the EERMA Tutor, EERMA Template Generator, and EERMA CASE Interface. The tutor is a hypertext-based system written in KnowledgePro, a product of Knowledge Garden, Inc. EERMA Template Generator, written in Arity Prolog, generates EERMA template sentences and lets the user fill in blanks to describe a data model. It creates an ASCII file containing the user's English template sentences. EERMA CASE Tool, written in Prolog, reads the output of EERMA Template Generator and passes it to a commercially available CASE tool which, in turn, creates an EER diagram.

6 Conclusion

In this paper, we discussed the design of a database design tool and various issues in implementing the tool using VP-Expert and dBase III+. VP-Expert was used to validate the design process, and dBase was used to record the database structure. We chose dBase because VP-Expert did not allow us to keep the database structure internally. VP-Expert could, however, append, modify, and search dBase files. Alternatively, the structure could have been saved to an ASCII file, but searching would be more difficult and require more programming than relying on a database system. The nature of this project was highly procedural and forward-chaining oriented since the final
database structure is incrementally built as the system evokes responses from a user. This required the use of a combination of backward chaining and forward chaining, one dimensional arrays, nested looping, and database access statements, fully exploiting the various features of VP-Expert. Throughout this project, in addition to typical rule-based programming, we needed and developed systematic, general skills to handle complex procedural aspects and to accommodate the forward chaining nature. Since the control mechanism techniques using rules have been established, additional modules and rules for relationships, abstractions and existence dependency can be readily added, which is a major advantage of using an ES shell. Most importantly though, a systematic, general approach to programming procedural-like design applications in ES shells was successfully developed using a purely rule-based ES shell. The specific VP-Expert problems in this project were summarized in Section 4.4.

A comparison was presented between conventional programming and programming declarative shells in a procedural manner. Our experience throughout this project indicates that writing complex applications using ES shells needs many aspects of conventional programming languages such as semantically clear statements, a flexible control mechanism, and facility for the logical grouping of related statements. From this comparison, it was determined that by applying a systematic, structured approach, properly commenting the code, and logically organizing rules, ES programming is a viable alternative to conventional systems for procedural design applications like DBD. Still, from the struggle with VP-Expert, a large powerful, hybrid, embedded shell would be the better tool.

References

Appendix:
Sample Segments of Code in VP-Expert

```
************** INITIALIZATION **************
MOUSEOFF
    MENU pick_ent,entity_val <> entity,attlist,entity_val
RESET all
    DISPLAY "This system will verify the enterprise (hit ENTER)"
    ! FIND enterprise ! THIS COULD BE USED FOR THE FILENAME
    replace_e1_check = true
    replace_e2_check = true
    entity_start = true
    ent_real_att = false
    last_att_q = false
    change_q_att = false
    pick_ent = false
GET all, entity_f, entity ! FIRST RECORD IS FILE NAME
j = 1
    DISPLAY "Processing, please wait"
```
Checking for repetitive items ...

* FIRST LOOP TO GET ENTITIES IN DIMENSIONED VARIABLES
WHILE KNOWN entity
  GET all, entity_f, ALL
  ent[i] = (entity)
  ent_id[i] = (id)
  i = (i + 1)
END
num_ent = (i - 1)

*]

* LOOP FOR ENTITY REPETITION **********
RESET i
  close entity_f
  GET all, entity_f, entity
  FOR i = 1 to @num_ent
    entity_1 = (ent[i]) ! GET FIRST ENTITY VALUE
    id_1 = (ent_id[i])
    FIND looping
    RESET looping
    CLOSE entity_f
    GET all, entity_f, entity
  END

*** LOOPING RULES *********************
RULE first_loop! REPEITIVE ENTITY NAMES
IF entity_start = true ! ALWAYS TRUE
  THEN looping = true
! CHECK ONLY AFTER ENTITY_1
  FOR ii = 1 to @i
    GET all, entity_f, entity
  end
  GET all, entity_f, all

* BACKWARD CHAINING RULES FOR REPEITIVE ENTITY NAMES  
SAME ENTITY FOUND - SHOULD YOU DELETE
RULE check_if_should_delete
IF same_entity = true
  THEN delete_check = true
  DISPLAY "Two items are called (entity_1). Should one be deleted?"
  DISPLAY "All properties of (entity_1) will be saved."
  ASK DELETE QUESTION
  RESET delete_q
  FIND delete_q;

*** FORWARD CHAINING FOR REPEITIVE ENTITY NAMES **********
USER WANTS TO DELETE (HOW TO USE NULLS??)
WHENEVER delete_q=yes
  IF delete_q = yes
    THEN CLOSE entity_f
    GET entity_1 = entity, entity_f, entity
    ! had this wrong
    GET entity_1 = entity, entity_f, all ! DELETE ENTITY_2
    entity = NONE ! USED TO INDICATE NULL VALUE
    PUT entity_f;

*** VARIOUS USER INTERFACE QUESTIONS *****
ASK enterprise: "What is the entity file you are working with?";
ASK delete_q: "Delete the entity (entity_1)?";
CHOICES delete_q: yes, no;
ASK relate_q: "Are you trying to relate these items?"
  ;
  CHOICES relate_q: yes, no;