# Building Usable Menu-Based Natural Language Interfaces To Databases

Craig W. Thompson, Kenneth M. Ross,
Harry R. Tennant and Richard M. Saenz

Central Research Laboratories
Texas Instruments Incorporated
Dallas, Texas

ABSTRACT. Natural language interfaces to databases are not in common use today for two main reasons: they are difficult to use and they are expensive to build and maintain. This paper presents a functional overview of a new kind of natural language interface that goes far in overcoming both of these problems. The "ease-of-use" problem is solved by wedding a menu-based interaction technique to a traditional semantic grammar-driven natural language system. Using this approach, all user queries are "understood" by the system. The "creation and maintenance problem" is solved by designing a core grammar with parameters supplied by the data dictionary and then automatically generating semantic grammars covering some selected subpart of the user's data. Automatically generated natural language interfaces offer the user an attractive way to group semantically related tables together, to model a user's access rights, and to model a user's view of supported joins paths in a database.

## 1.0 INTRODUCTION

One major goal of much work done in AI and computational linguistics in the last 15 years has been to make natural language interfaces to software that naive users could use. Naive users might be new users or occasional users or users who don't want to use a formal query language. The motivation has been that people know a natural language and won't forget how to ask questions in it. The assumption has been that the burden is on the computer to "understand" the user's naturally phrased query or command and take some appropriate action. Most natural language interface work has targetted on database systems. Yet, only a few commercial natural language interfaces are available today, including the Intellect system by Larry Harris of AI Corporation, Straight Talk by Gary Hendrix of Symantec, and Pearl by Roger Schank of Cognitive Systems. Two good reasons explain why: first, existing systems are brittle and users are often frustrated in using them; and second, natural language interfaces are expensive to build and maintain.

This remainder of this paper is structured as follows: The rest of section 1.0 motivates and describes the general approach we have taken to solving the two problems with existing natural language systems. Section 2.0 describes the architecture of the implemented system at a functional level. Section 3.0 discusses the advantages and limitations of the approaches taken here.

### 1.1 A SOLUTION TO THE 'USABILITY' PROBLEM

In one of the few evaluations of a natural language interface system, Tennant 1980 found that a major problem with PLANES was that, even after a training session where the capabilities and limitations of the system were explained to users, users quickly developed negative expectations concerning what the linguistic and semantic coverage of the system was. That is, because PLANES had a one third error rate on even simple queries, users did not ask more complex queries, many of which could be handled by PLANES. Tennant also found that users were unable to distinguish between the limitations in the system's conceptual coverage and the system's linguistic coverage. Users did not successfully adapt to the system's limitations after some amount of use because there was no clear path that naive users could use to learn these limits. Problems in using PLANES rendered users unable to successfully solve many of the problems they were assigned as part of the evaluation of PLANES, even though these problems had been specifically designed to correspond to some relatively straightforward queries that PLANES could understand. These inferences about

PLANES' capabilities resulted in much user frustration because of their very limited assumptions about what PLANES could understand. The coverage mismatch problem pointed out by Tennant is a general problem that must be faced by any natural language interface.

There are three traditional approaches to solving the coverage mismatch problems mentioned above. The first is a set of training sessions to teach the user the syntax and conceptual coverage of the system. Interestingly, users of Harris' INTELLECT system are told that certain words (like LIST and GROUP BY) are keywords. However, there are several problems with this approach. First, it does not allow untrained novices to use such a system. Second, it assumes that infrequent users will remember what they learn about the coverage of the system. Third, training sessions can only give the user a partial idea of coverage. The second approach to solving the coverage mismatch problem is to extend the coverage of the system to the point where practically all inputs are understood. By doing this, most sentences that are input will be understood and few negative expectations will be created for the user. In natural language interfaces, the design methodology has often been to trap users' queries that could not be interpreted by the system, analyze them, and then add capabilities to the system to cover the input. Unfortunately, this methodology often results in spotty coverage, so that a similar input may not be covered and users have trouble generalizing what is and what is not in the scope of the language. So this approach can actually contribute to the problem of allowing a user to generalize about the coverage of the system. The design goal of covering every user input has the additional disadvantage of being very open-ended. Large grammars result and there is no clear criteria to tell the system implementer when he is done building the interface. The third approach to solving the coverage mismatch problem is to engage the user in a "clarification dialogue" whenever his query is ambiguous, incomplete, or otherwise outside of the bounds of the coverage of the system. This approach was explored in Codd's Rendezvous Codd 1978 . Here too, there are problems. Clarification dialogues require the user to read restatements of their query and users have some trouble comparing restatements with their original phrasing.

In this paper, we will apply a technique that uses current technology (current grammar formalisms, parsing techniques, etc.) to make natural language interface systems meet the criteria of usability by novice users. To do this, user expectations must closely match system performance. Thus, the interface system must somehow make clear to the user what the coverage of the system is.

The NLMENU System, described in this paper, is a grammar-driven menu-based natural language interface system. Rather than requiring the user to type his input to the natural language understanding system, he is presented with a constellation of menus on the upper half of a high resolution bit map display. Sample screens for the NLMENU system are included at the end of the paper. (See Figures 1-3). Using an interaction technique of his choice (a mouse, speaker-dependent speech, keyboard commands, or typing), the user can choose the words and phrases that make up his command or query. The user chooses items from "active menus", which are highlighted in the figures. As he chooses items, they are inserted into the 'sentence' window on the lower half of the screen. Some sample sentences follow:

Find the average weight of parts whose part color is red or blue and which are supplied by suppliers whose supplier status is greater than 10.
Find course# and description of courses taught by instructors named Thompson or Ross and whose prerequisites are courses whose course title is Structured Programming.
Delete parts whose part status is between 10 and 20.

As a sentence is constructed, the active menus and items in them change to reflect only the grammatically legal choices, given the portion of the sentence that has already been input. At any point in the construction of a natural language sentence, only those words or phrases that could legally come next are displayed in an active menu for the user to select. Thus, sentences which cannot be processed by the natural language system can never be input to the system. By retaining both active and inactive menus in the display, both the scope and limitations of the system are made clear to the user. Thus, the set of statable queries exactly defines the linguistic and conceptual coverage of the system. This approach solves many of the problems having to do with "ease-of-use" of natural language interfaces.

## 1.2 A SOLUTION TO THE 'PORTABILITY' PROBLEM

This paper also contributes to the solution of the second problem, of making natural language interfaces easy to build and maintain, in the very important special case of relational databases and in the context of a grammar-driven, menu-based interface driver. In this context, this paper addresses the following problems: existing natural language interfaces to databases are seldom portable; most are application-specific. They take from man-weeks to man-years for a specialist to build. They are not robust with regard to changes in the data they are interfacing to. They are hard to debug. And there is no established way to guarantee that they cover the desired data or

the functionality of the target computer system. So, using existing approaches, natural language interfaces to databases will be built only for important database applications. Applications must justify the expense.

Section 2.2 describes an implemented system which automatically generates natural language interfaces to relational databases. The interfaces are for use with an NLMENU grammar-driven, menu-based system. The basic idea is that domain specific parameters are elicited interactively from a user and then substituted into a domain independent natural language core grammar and corresponding lexicon and a semantic grammar and lexicon result. Together, the semantic grammar and lexicon define a natural language interface to some semantically related set of database tables.

Interfaces that have been built with the techniques described here include versions of well-known experimental natural language interfaces PLANES, LUNAR, LADDER, TQA, RENDEZVOUS, and INTELLECT/EMPLOYEES (original data is only available for LADDER). In addition, we have built NLMENU interfaces to several TI Internal NLMENU databases; some personal databases like MY CONFERENCES and MY CITATIONS; some toy databases like Date's SUPPLIER-PARTS database Date, 1981 , a JOBSHOP database, a BASEBALL database and a UNIVERSITY database; and the SYSTEM RELATIONS database.

In the past several years a number of researchers have been interested in the portability issue. Kaplan 1979 , Harris 1979 , Hendrix and Lewis 1981 , and Grosz et al 1982 all provide insights into some aspects of portability. Kaplan describes a portable system in which an expert can port to a new domain in a matter of hours. Harris' Intellect has been ported to a variety of applications. It takes a system person a day to build a bare bones interface and a month is needed to reach a finished product. Both Hendrix and Grosz describe a prototype system, first called Ted and later Team, which allows a database expert who is not necessarily a natural language expert to build new interface. They describe an acquisition dialogue in which the designer interactively specifies lexical information (synonyms, antonyms, verb conjugations, +-human), and also database structural information (like what attributes are numeric). This information provides parameters to a pragmatic grammar. None of the papers above give the reader any real insight into how expert a user had to be to build an interface, how long it took to build one, whether it was easier to build some interfaces than others, and whether the resulting interfaces were usable.

Our work differs from past work in two ways: first, we concentrate on crafting a small, expressive, carefuly designed core grammar and lexicon. We provide a guided path towards expressing a query, but not a general paraphrasing capability. The grammars and lexicons produced by the "Build Interfaces" interface (see below) are for use with an NLMENU system and would be very inadequate in traditional systems. The principal reason is that they are purposely engineered to provide only a limited set of grammatical and lexical ways of expressing a statement. They are aimed at taking advantage of a person's ability to understand a fragment of natural language written in a limited language and at guiding him to express himself in that limited language. There is no intent to cover any more natural language than a domain requires, so the problem of building an interface is not open-ended. Second, end-users can build interfaces in a short period of time without needing to become familiar with grammars and lexicons. The interface specification dialogue itself is NLMENU driven. It makes no use of linguistic information but makes heavy use of the data dictionary. In the simplest case, a user can build a new interface simply by choosing from a menu a set of tables that he wants the interface to cover. Automatically generated interfaces are quite usable: people who have never seen a lisp machine before can formulate interesting queries using automatically generated natural language interfaces, as often happens in our demos.

## 2.0 SYSTEM ARCHITECTURE

The system described in this paper was prototyped on LMI lisp machines. The prototype served as a specification for a commercial product, written in C, which will be available on the 8088-based TI Professional Computer and which interfaces the menu-driven natural language interface technology to Oracle's SQL. The prototype NLMENU system is implemented in Lisp Machine Lisp and consists of the following software components: a window management system, a target lisp machine relational dbms, a parser, an NLMENU driver, a General Sessioner and a set of NLMENU driven interfaces including various natural language interfaces, a GUIDED SQL interface, and a BUILD INTERFACE interface. Each NLMENU-driven interface consists of a grammar, a lexicon, a set of experts, and possibly a target software system. This section describes each of the components in turn. The significance of the more important components is discussed in section 3.0.

### 2.1 THE BASIC NLMENU SYSTEM ARCHITECTURE

The WINDOW SYSTEM, including the menu subsystem, is fully described in documentation from Lisp Machines Incorporated. It is based on "flavors", an object-oriented, hierarchical data structure with message-passing that is available in newer Lisps. The window system contains

primitives for building various kinds of menus and for building constraint frames of menus and windows like the ones in NLMENU screens.

Two target RELATIONAL DBMS's have been interfaced to in the prototype NLMENU system: Oracle's implementation of SQL and a prototype relational dbms on the lisp machines, which uses a relational algebra.

The PARSER is a "modified" left-corner, bottom up, all paths, attributed grammar parser Ross 1983 . The modifications enable the parser to parse a menu item (word or phrase) at a time and to predict the set of next possible words in a sentence, given the input that has come before. The grammars employed in the NLMENU system are semantic grammars Burton, 1976 written in a context-free grammar formalism. Translation of the sentence is done as the sentence is parsed, using lambda conversion. Translations are associated with each of the words and phrases in the lexicon. Associated with each context-free rule is a semantic rule indicating the order in which the translations of the nodes to the right of the arrow are to be combined.

The NLMENU DRIVER is an input loop which accepts user's input (in the form of a menu choice). If the menu choice comes from the "commands" menu, one of the following actions is taken:

- RE-START--reinitialize the screen for another query
- RUBOUT--rubout the last menu choice from the end of the current sentence being composed
- SHOW QUERY--when a completed sentence has been entered, the translation of the query or command into the database query language is displayed in the output window (see Fig 2)
- EXECUTE--the query is executed and the result displayed in the output window (see Fig 3)
- EDIT ITEM--a mode in which the owner of an interface can rephrase awkwardly phrased automatically generated menu items
- EXIT--exit the driver, leaving it in the current state, in case the user later returns to the interface
- SAVE Q, RETRIEVE Q, DELETE Q, PLAY Q--queries can be saved, recalled (or deleted) from a menu of saved queries, or a set of queries can be "played" automatically for demo purposes.

If the menu choice is one of the active menu items, the driver parses that choice and then predicts the set of next legal grammar terminals. It then refreshes a display of next legal choices and the user chooses one.

The NLMENU system does not store the words that correspond to data items in the lexicon as many other natural language systems do. Instead, a meta category called an EXPERT is stored in the lexicon. As an example, when a user's sentence is "Find parts whose part color is ...", a PART_COLOR_EXPERT pops up a menu of legal part colors. An expert is an arbitrary procedure which the user may supply but which defaults to some system supplied procedure. Three default procedures that are particularly useful are: present the user with a menu of data items chosen from a closed semantic domain, or go directly to the database and populate a menu from the projection of an attribute, or simply allow the user to type in a value and use the data dictionary to validate the value the user types. This last sort of expert is particularly useful when the database is remote and it is undesirable to execute sub-queries while the sentence is being built. Experts may be much more exotic: In our example above, on a color monitor, an expert could pop up a color chart and let the user choose a color from it. When interfacing to our spatial database, we implemented an expert that allowed a user to pick a latitude/longitude rectangle off of a map to specify an area.

Many systems allow ELLIPSIS to permit the user to, in effect, ask a parameterized query. For example, in Ladder, a query like "Find ships whose speed is greater than 50 knots and which are in the Mediterannean" might be followed by typing "30 knots", which has the effect of re-running the query with the new parameter. In our system, we handle ellipsis in a more immediate way, by structure editing. To change a "parameter", we simply move the mouse to a phrase generated by an expert and select that item. The expert which originally produced that item is then called, allowing the user to change that item to something else. Our approach gets around problems of elliptical ambiguity as in "Find ships whose status is 10 and whose speed is over 30" followed by "20".

Since the natural language semantic grammar is technically unrestrictedly context free and a subset of English, ambiguous sentences can be created. In the NLMENU system, by design, lexical ambiguity (where one lexical item from a given syntactic category has two or more translations) does not occur. But structural ambiguity can occur. In our system, if a user tries to execute an ambiguous query, the system offers him a menu of possible interpretations. The interpretations are distinguished by indentation and numbering, as in:

```
"Find courses which are prerequisites of
   courses (1) whose course department is
               Computer Science
       and (2) whose course credits is 3."

"Find courses (1) which are prerequisites of
                  courses whose course
                  department is Computer Science
          and (2) whose course credits is 3."
```

This simple approach contrasts with the standard solution of natural language systems which is to

paraphrase a user's ambiguous query. That approach requires a paraphraser module and also requires a user to look at multiple paraphrases, and people often have trouble choosing the interpretation they want.

One interesting note about our grammar is worth mentioning. In English, there is no really algorithmic way to decide what the user means when he uses conjunction and disjunction together. Possible implementations might include left-to-right parsing with AND and OR of equal precedence, some AND/OR precedence rule, some heuristic approach or a hybrid. If more than one approach is taken, rampant ambiguity results. One can always find contradictions to a heuristic approach. We finally settled on the AND/OR precedence approach found in many programming languages (we even allow parenthesis to override precedence), because programmer/users are already familiar with the idea and it is not hard to learn. We performed a human factors experiment to verify that this approach was reasonable and the results bore out our conjecture that people can easily learn to use the feature. In addition, we found it desirable to include a reference to the thing modified in modifying phrases (as in "whose COURSE department is"). Although stilted, the English is readily understood and ambiguities like deciding whether "whose department is" modifies INSTRUCTORS or COURSES can be avoided.

In addition to the above software modules, a HELP SYSTEM is available for users. At any point in a query, a user can get help on a menu item or a menu itself; he can use mouse buttons or the keyboard to make his request. For automatically generated interfaces, the "help message" can be automatically generated. A message about an attribute may include its documentation, its range if restricted, its units if any, its format if any, etc. Help on active menu items also displays the set of active items that would be available if the item were chosen. As with menu items themselves, automatically generated help messages may be edited by the user.

The GENERAL SESSIONER module (see Figure 4) is a top-level driver that checks a users password, and then presents him with a menu which gives him choices between system commands, user-owned natural language interfaces (those that the user created), interfaces granted to the user, and interfaces granted to the PUBLIC. Naturally, different users see different menus according to their access rights to various NLMENU interfaces. Two system-owned relations:

NLMENU-INTERFACES(owner, interface_name, target-dbms, portable-spec, grammar, lexicon, window-description)

NLMENU-GRANTS(owner, interface-name, user)

govern which interfaces users own and which interfaces users have been granted access rights to.

The GUIDED SQL choice on the general sessioner menu allows a user to use the NLMENU driver with a formal SQL grammar. Such a grammar is not a semantic grammar in the sense of the natural language grammars--that is, constraints governing what relations and attributes can fill identifier roles are not necessarily satisfied as they are in the natural language NLMENU grammars. But, by using the GUIDED SQL interface, users can be guaranteed of making no syntactic errors in specifying database queries or requests. This interface is just representative of a menu-based grammar-driven interface to any formal language, by no means restricted to database query languages.

## 2.2 AUTOMATICALLY GENERATING A NATURAL LANGUAGE INTERFACE

This section discusses how an end-user can build his own natural language interface to data that he owns or has been granted access to. The user needs no knowledge of grammars, lexicons, the target query language, etc., but only an elementary knowledge of tables, keys and joins. So a large class of users can build their own interfaces. First, the BUILD INTERFACES interface is discussed and operations on interfaces are described. Then, the CREATE and MODIFY operations are described as a means of eliciting domain-dependent customization and coverage parameters from the user. These parameters are stored in a data structure called a "portable spec". Finally, the method whereby a semantic grammar and lexicon are generated from a core grammar and a portable spec is discussed.

The BUILD INTERFACES module (see Figure 5) is an NLMENU driven interface consisting of a grammar, lexicon, window description, and an underlying semantics which defines the following operations:

TUTORIAL--an on-line tutorial on the BUILD INTERFACES interface
LIST INTERFACES--list interfaces owned or granted to the user
CREATE INTERFACE--create a new NLMENU interface covering a set of tables
MODIFY INTERFACE--modify an existing owned NLMENU interface
COMBINE INTERFACES--merge two interfaces
GRANT INTERFACE--grant owned interface(s) to other user(s)
REVOKE INTERFACE--revoke a granted interface
DROP INTERFACE--drop owned interfaces

Each of the commands has a simple English-like syntax. An effort was made to make the keyword phrasing of the commands compatible with SQL,

our usual target query language.

The CREATE INTERFACE and MODIFY INTERFACE commands are the heart of BUILD INTERFACES. Both commands operate on a (new or existing) domain specific data structure called a PORTABLE SPEC and interactively allow a user to fill in slots in the structure. A portable spec consists of a list of categories. The categories are as follows: the COVERED TABLES list specifies all relations or views that the interface will cover. The retrieval, insertion, deletion and modification relations specify ACCESS RIGHTS on selected covered tables. Non-numeric attributes, numeric attributes and computable attributes CLASSIFY ATTRIBUTES according to type. Computable attributes are numeric attributes that are averageable, summable, etc. A user may also choose not to cover some attributes in an interface. IDENTIFYING ATTRIBUTES are attributes that can be used to identify the rows. Typically, identifying attributes will include the key attributes, but may include other attributes if they better identify tuples (rows) or may even not include a full key if one seeks to identify sets of rows together. TWO TABLE JOINS specify supported join paths between tables. THREE TABLE JOINS specify supported "relationships" (in the entity-relationship data model sense) where one relation relates 2 others. The TABLE, ATTRIBUTE and INSERTION EXPERTS define user supplied expert definitions to replace system defaults. EDITED ITEMS provides a list of old and new phrasings of menu items. And the EDITED HELP provides a way for users to add to, modify, or replace automatically generated help messages.

Popup expert menus guarantee that the user will choose only from legal choices when selecting parameter values. Categories COVERED TABLES, ACCESS RIGHTS, CLASSIFY ATTRIBUTES, IDENTIFYING ATTRIBUTES, and TABLE JOINS all involve consulting the database data dictionary and then popping up various kinds of menus in which a user selects from legal options. Unspecified options are defaulted.

Some of the categories in the portable spec are best specified after the interface builder has created the interface. At that time, he can replace menu items or help messages with customized paraphrases. All such changes are recorded in the portable spec in case the interface is later modified. An interface resulting from a BUILD INTERFACE session is guaranteed to be valid in a sense described below.

### 2.2.1 AUTOMATICALLY GENERATING A NATURAL LANGUAGE INTERFACE FROM A PORTABLE SPEC

The function MAKE-PORTABLE-INTERFACE takes as input a portable spec, uses it to instantiate a domain independent core grammar and lexicon, and returns a semantic grammar and a semantic lexicon pair, which defines an NLMENU interface.

A portable spec data structure is the input to both a MAKE-SEMANTIC-GRAMMAR and a MAKE-SEMANTIC-LEXICON routine to be described. These routines do not verify the integrity of specs though they could easily be modified to do so. Instead, it is assumed that the component that provides the parameters has done this validation. This is guaranteed to be the case when a portable spec is specified using the BUILD INTERFACES interaction.

The function MAKE-SEMANTIC-GRAMMAR is defined as follows:

```
MAKE-SEMANTIC-GRAMMAR(portable-spec) -->
  semantic-grammar.
```

Grammar rules have two parts: a context free rule part and an interpretation part telling how to combine translations associated with the elements on the right hand side of the grammar rule to make a translation to associate with the element on the left hand side of the grammar rule. The basic operation of the MAKE-SEMANTIC-GRAMMAR function is identifier substitution. Generally this occurs in a context of looping through one of the portable spec categories, say non-numeric-attributes, and substituting every relation and attribute pair into a given rule template. So given the rule template:

```
⟨rel⟩-mod  -->  whose-⟨rel⟩-⟨attr⟩-is
                ⟨rel⟩-⟨attr⟩-expert  ((1 2)),
```

```
if non-numeric-attributes =
   ((PART city color name part#)
    (SUPPLIER city name supplier#)
    (SHIPMENT part# supplier#))
```

then 9 grammar rules will result. The first will be:

```
PART-mod  -->  whose-PART-CITY-is
               PART-CITY-expert  ((1 2))
```

Function MAKE-SEMANTIC-LEXICON works analogously:

```
MAKE-SEMANTIC-LEXICON(portable-spec)  -->
  semantic-lexicon.
```

Here each form being substituted into results in a LEXICAL ENTRY consisting of a 5-tuple with fields ⟨category, menu-item, menu-window, translation, help-text⟩. The category corresponds to a terminal element in the grammar (that is, it appears on the right hand side, but not on the left hand side, of one or more grammar rules). The menu-item is a string (word or phrase or whatever) to display as an item in some menu-window. The menu-window identifies in which pane a menu-item will appear. The

translation lists a fragment of code written in the target software system. Whenever interfacing to a new target database system, only this portion need be re-written. At present we have translations which map natural language to our lisp machine relational dbms and to IBM's SQL. An example of an instantiated lexical rule for our example is:

```
(whose-PART-CITY-is
 "whose part city is"
 modifiers
 (LAMBDA Y (%% (RETRIEVE 'PART
   WHERE (MEMBER CITY 'Y))))
 "The CITY attribute of relation PART has the
  following documentation:
     the city a part is in at the moment
  and comes from the SUPPLIER-CITIES semantic
  domain, which is an ordered set of
  .'Paris', 'London', 'Rome', 'New York' ")
```

The core grammar and lexicon can be small (on the order of 25 grammar rules and 40 lexical entries), but the size of the resulting semantic grammars and lexicons will depend on the portable spec. (72 semantic grammar rules and 84 lexical entries result from instantiating the core grammar and lexicon with the portable spec that describes the 3 relations in the supplier-parts database from Date, 1982 :

```
SUPPLIER(supplier# name city status)
PART(part# name city color weight)
SHIPMENT(supplier# part# quantity)
```

Since substitution is uniform, no rules can be carelessly excluded. So all the tables and their attributes will be covered. The next section describes an algorithm that checks the well-formedness of generated grammars and lexicons.

### 2.2.2 WELL-FORMEDNESS TESTING AND VALIDATION.

The function

```
(WELL-FORMEDNESS-TEST nlmenu-grammar
                      nlmenu-lexicon)
```

invokes a static collection of tests to find bugs in either an automatically generated NLMENU grammar and lexicon pair or a manually-generated one. The function finds the following problems:

- o unreachable grammar non-terminals
- o items that are both non-terminals and lexical categories.
- o unused lexical items: these are in the lexicon but are not grammar leaves.
- o undefined lexical items: these appear as leaves in the grammar but are not in the lexicon.

This test is clearly useful for manually generated NLMENU interfaces, but it is also useful for testing and debugging changes and additions made to core grammars and lexicons.

In addition to finding bugs, the test can be used at grammar-lexicon writing time: One of the values returned by WELL-FORMEDNESS-TEST (grammar, nil) is a list of all lexical categories that the grammar writer must write lexical entries for. The WELL-FORMEDNESS-TEST was used in the development of a GUIDED SQL interface as well as in debugging several core natural language grammar and lexicon pairs.

The function (VALIDATE spec) checks to make sure that a portable spec data structure is well-formed and reflects an existing data dictionary state. The categories of the spec are verified against the data dictionary where the definitions of tables are stored. VALIDATE checks that specified relations and views really are tables in the database and that the user has the access rights reflected in the categories RETRIEVAL RIGHTS, INSERTION RIGHTS, etc., checks to make sure the attributes are classified correctly according to types non-numeric or numeric, checks that at least a candidate key of the relation is a (possibly proper) subset of the identifying attributes, and checks the join fields to make sure they are of the same (or comparable) semantic data type. For rich data dictionaries, all this can be supported. For more impoverished ones, like SQL's, less checking can be provided. For instance, since the only data types supported (until recently) are CHAR and NUM there can be no guarantees provided by the system that joins are over semantically compatible domains. In our implementation, the validate function is replaced by an interactive component which elicits only valid information reflecting the current database data dictionary state.

An interface is provably correct if the spec is valid and the core grammar and lexicon are correct. The proof that core grammar and lexicon covers a target underlying software system requires arguing along the following lines: functionality in the target language is identified and then natural language constructions are identified that translate to those identified target functions. After verifying coverage, the well-formedness test can be applied to show that the core grammar and lexicon are well-formed. No proof of naturalness of an interface language is possible; the naturalness of the interface language can only be ascertained by human factors testing or by reference to known results of human factors tests.

## 3.0 ADVANTAGES AND LIMITATIONS

The menu approach to natural language input has many advantages over the traditional typing approach. Most importantly, every sentence that is input is understood. The fact that the menu-based natural language understanding systems guide the user to the input he desires

is beneficial for two other reasons. First, confused users who don't know how to formulate their input need not compose their input in a vacuum. They only need to recognize their input by looking at the menus. Second, the extent of the system's conceptual coverage will be apparent. The user will immediately know what the system knows about and what it does not know about.

Some advantages accrue because the grammars required can be small. First, implementation time is greatly decreased. Generally, writing a thorough grammar for an application of a natural language understanding system consumes most of the development time. Second, it has also proved to be feasible to put the NLMENU System on a microcomputer. Third, parse time is small, since parse time is a function of grammar size.

Several questions arise with respect to a menu-based approach to building natural language interfaces. First, can users successfully use an NLMENU interface in which they have only one way to state their query? We have run a series of pilot studies using Tennant's methodology for evaluating natural language understanding systems. All subjects were successfully able to solve all of their problems. Comments from subjects indicated that although the phrasing of a query is at times stilted, subjects were not bothered by this and could find the alternative phrasing without any difficulty.

A second question arises: Since the size of the lexicon determines the number of items that need to be displayed on an NLMENU screen, is menu size a problem? Menus must not become too big or the user will be swamped with choices and will be unable to find the right one. For most of the interfaces we have generated, this has not been a problem, since choices earlier in a sentence tend to restrict later choices to a manageable few. Only for interfaces with a large number of relations (over 10, say) or with relations with a large number of attributes (over 20, say) do 'recognition problems' start to occur. All our menus are scrollable. Other interaction techniques can be used to put off the problem. But eventually, menu size does limit the sort of interfaces one can use the NLMENU approach for.

The BUILD INTERFACES natural language interface generator described here enjoys several practical and theoretical advantages:

1) END-USERS can construct natural language interfaces to their own data in minutes, not weeks or years, and without the aid of a grammar specialist.

2) The interface builder can control coverage. He can decide to make an interface that covers only a semantically related subset of his tables. He can choose to include some attributes and hide other attributes so that they cannot be mentioned. He can choose to support various kinds of joins with natural language phrases. He can mirror the access rights of a user in his interface, so that the interface will allow him to insert, delete, and modify as well as just retrieve and only from those tables that he has the specified privileges on. Thus, interfaces are highly tunable and the term "semantic coverage" can be given precise definition.

3) Automatically generated natural language interfaces are robust with respect to database changes; interfaces are easy to change if the user adds or deletes tables or changes table descriptions. One need only modify the portable spec to reflect the changes and regenerate the interface.

4) Automatically generated NLMENU interfaces are guaranteed to be correct (bug-free). The BUILD INTERFACES interface (see section 2.2), in which users specify the parameters defining an interface, insures that parameters are valid (correspond to real tables, attributes, and domains). A well-formedness test detects bugs in semantic grammars and lexicons, so a core grammar and lexicon can be debugged easily. Once debugged, a core grammar and a valid spec can be combined and the resulting interface will be correct.

5) Natural language interfaces are constructed from semantically related tables that the user owns or has been granted and they reflect his access privileges (retrieval, insertion, etc). By extension, natural language interfaces become database objects in their own right. They are sharable (grantable and revokable) in a controlled way. A user can have several such NLMENU interfaces. Each gives him a user-view of a semantically related set of data. This notion of a view is like the notion of a database schema found in network and hierarchical but not relational systems. In relational systems, there is no convenient way for grouping tables together that are semantically related. Furthermore, an NLMENU interface can be treated as an object and can be GRANTed to other users, so a user acting as a database administrator can make NLMENU interfaces for classes of users too naive to build them themselves (like executives). Furthermore, interfaces can be combined by merging portable specs and so user's can combine different, related user-views if they wish. The ability to combine interfaces is also useful for incrementally building up a larger interface from a set of component interfaces.

6) Since an interface covers exactly and only the data and operations that the user chooses, it can be considered to be a "model of the user" in that it provide a well-bounded language that reflects a semantically related view of the

user's data and operations. Similarly, one can easily imagine a complicated language (like SQL) partitioned into a "ten statement SQL" core for novice users and a collection of add-on modules (for GRANTing or making INDEXes).

7) The last advantage is that even if an automatically generated interface is for some reason not quite what is needed for some application, it is much easier to first generate an interface this way and then modify it to suit specific needs than it is to build the entire interface by hand.

Taken together, the advantages listed above pave the way for low cost, maintainable, easy-to-use interfaces to relational database systems (and to a wide variety of other kinds of software as well). Many of the advantages are novel when considered with respect to past work. The significance of this work is that it makes it possible for a MUCH broader class of users and applications to use menu-based, natural language interfaces to databases.

Much work remains to be done. At present, we are beginning another round of human factors testing. And we are beginning to explore a number of features from traditional natural language approaches in the context of the NLMENU paradigm.

BIBLIOGRAPHY


Burton, Richard. "Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems". PhD Thesis, BBN Report #3453, BBN, Cambridge, MA, December, 1976.

Codd, E F, R S Arnold, J M Cadiou, C L Chang, and N Roussopoulos. "RENDEVOUS Version 1: An Experimental English Language Query Formulation System for Casual Users of Relational Databases". RJ2144 (29407), IBM San Jose, January 1978.

Date, C. J. An Introduction to Database Systems. (second edition, vol #1) Addison-Wesley, 1981.

Grosz, Barbara, Doug Appelt, Alex Archbold, Bob Morre, Gary Hendrix, Jerry Hobbs, Paul Martin, Jane Robinson, Daniel Sagalowicz, and Paul Warren. "TEAM: A Transportable Natural Language System". Technical Note 263, SRI International, Menlo Park, CA, April, 1982.

Harris, Larry. "Experience with ROBOT in 12 Commercial Natural Language Database Query Applications". Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan, August, 1979.

Hendrix, Gary and William Lewis. "Transportable Natural Language Interfaces to Databases". Proceedings of the 19th Association for Computational Linguistics, Stanford, June, 1981.

Kaplan, S Jerrold. "Cooperative Responses from a Portable Natural Language Query System". PhD Thesis, University of Pennsylvania, July 1979.

Ross, Kenneth. "An Improved Left-Corner Parsing Algorithm". Proceedings of COLING 82, 1982, pp 333-338.

Tennant, Harry R. "Evaluation of Natural Language Processors". PhD Thesis, Department of Computer Science, University of Illinois, 1980.

Tennant, H. R., K. M. Ross, R. M. Saenz, and C. W. Thompson. "Menu-Based Natural Language Understanding". 21st Annual Meeting of the Association for Computational Linguistics, MIT, June 15-17, 1983.

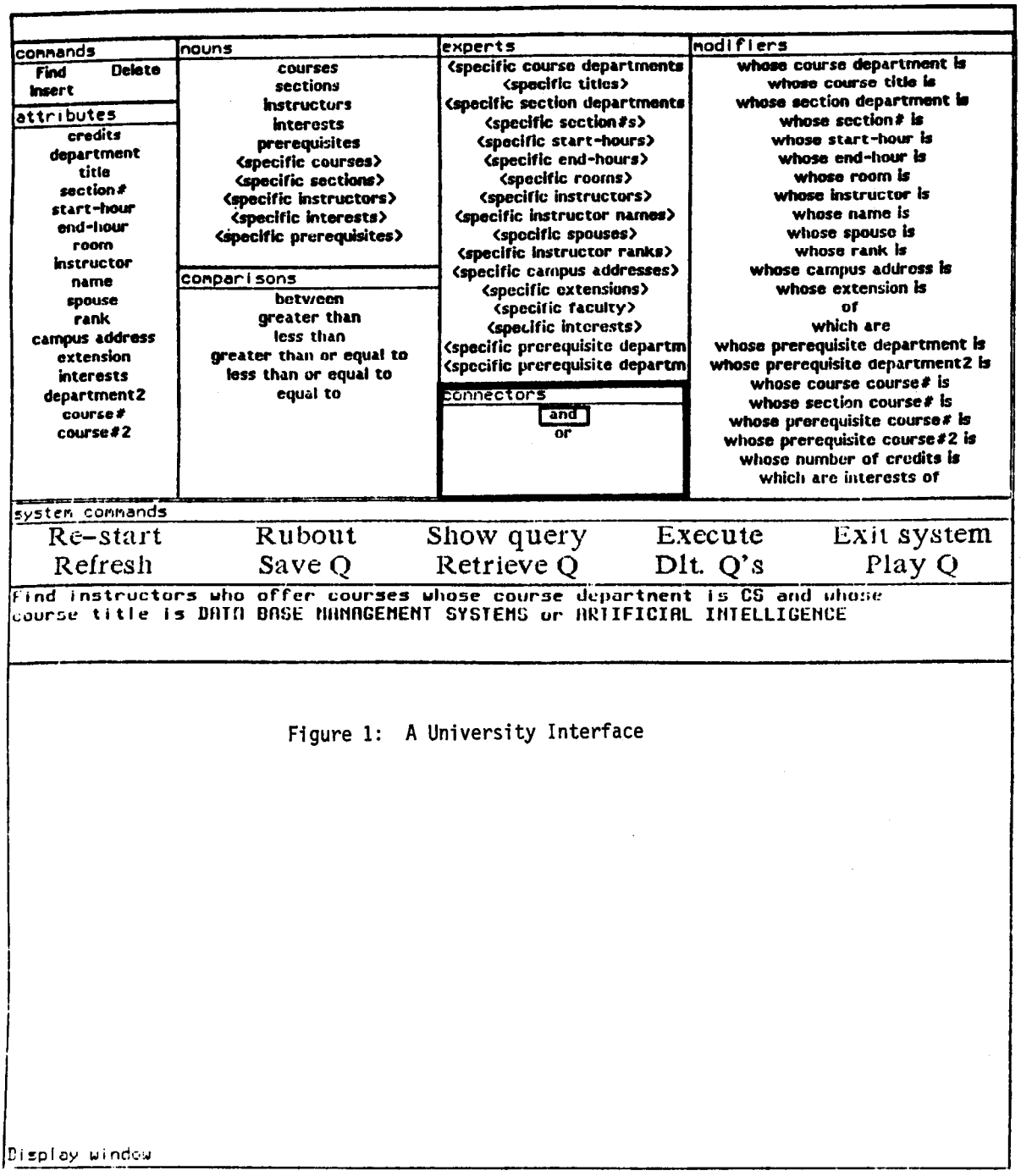


Figure 1: A University Interface

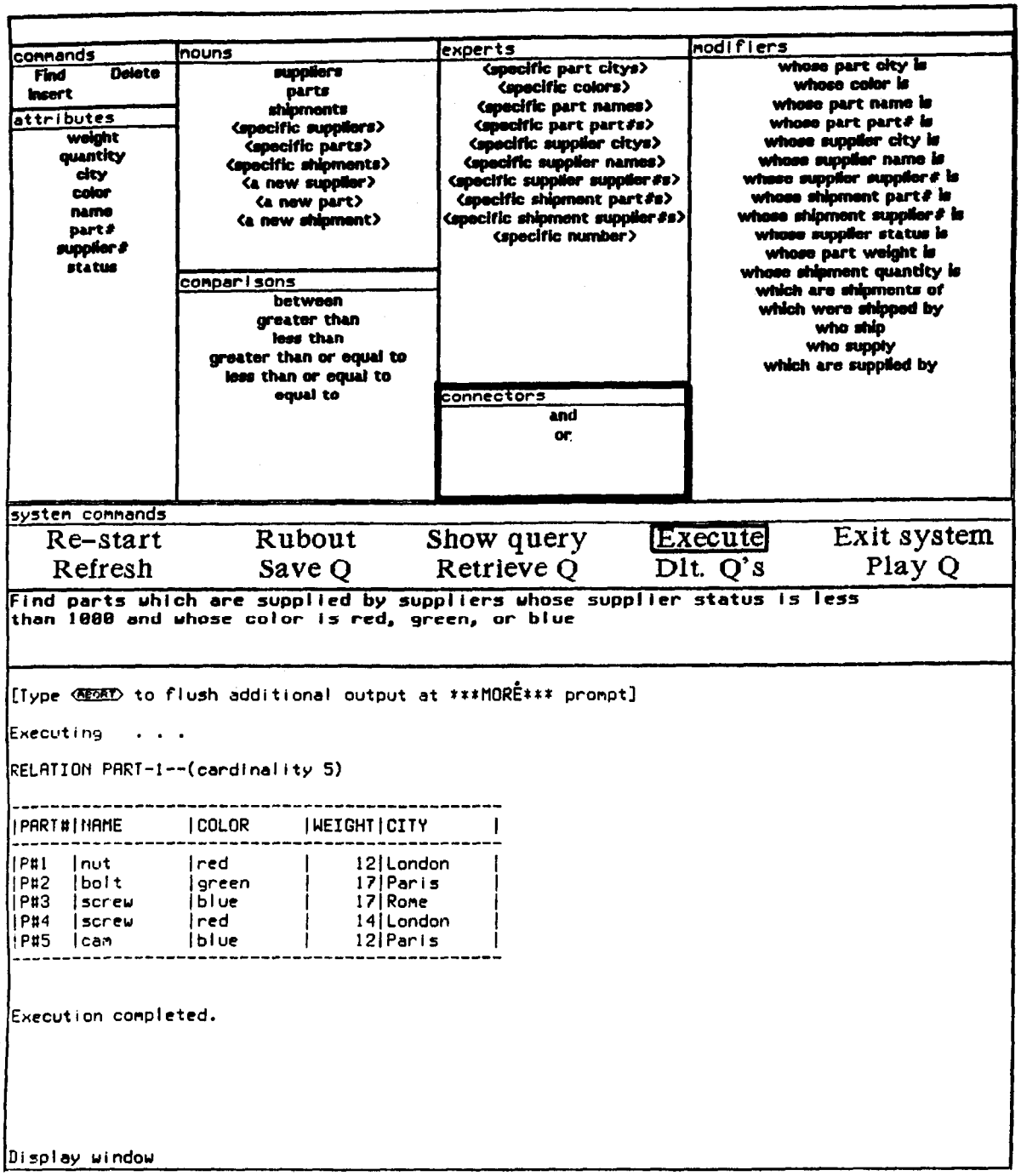


Figure 2: A Supplier-Parts Interface

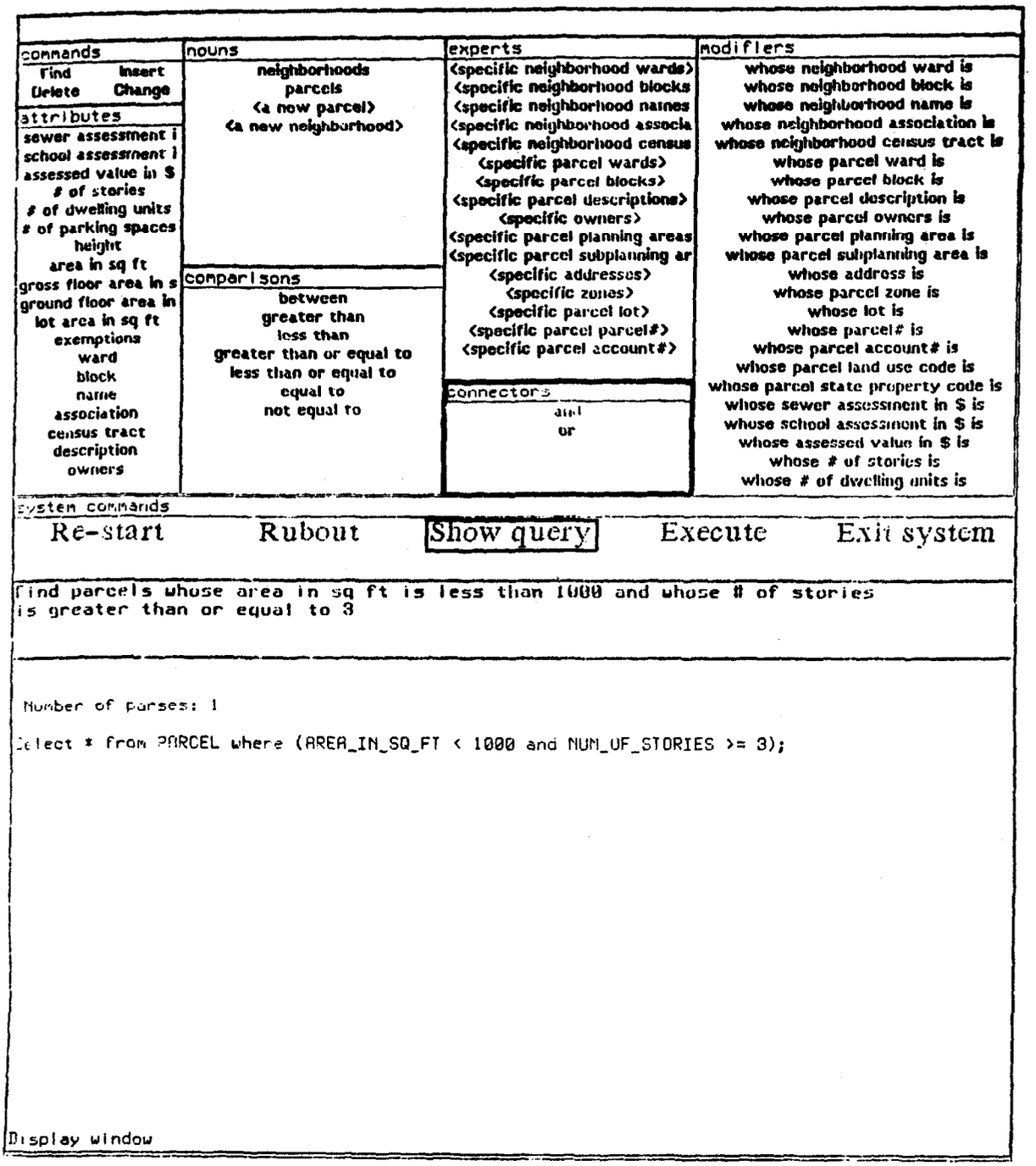


Figure 3: A TQA-like Interface

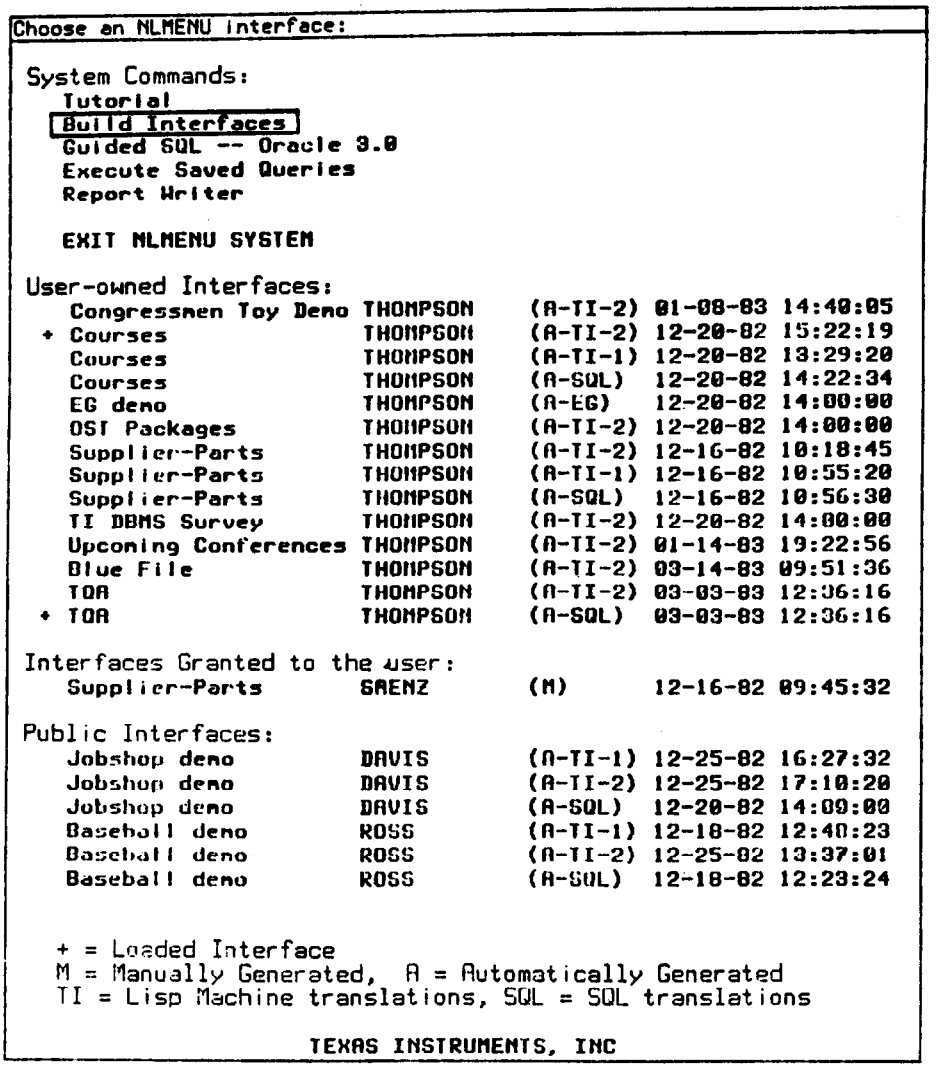


Figure 4: The General Sessioner Menu

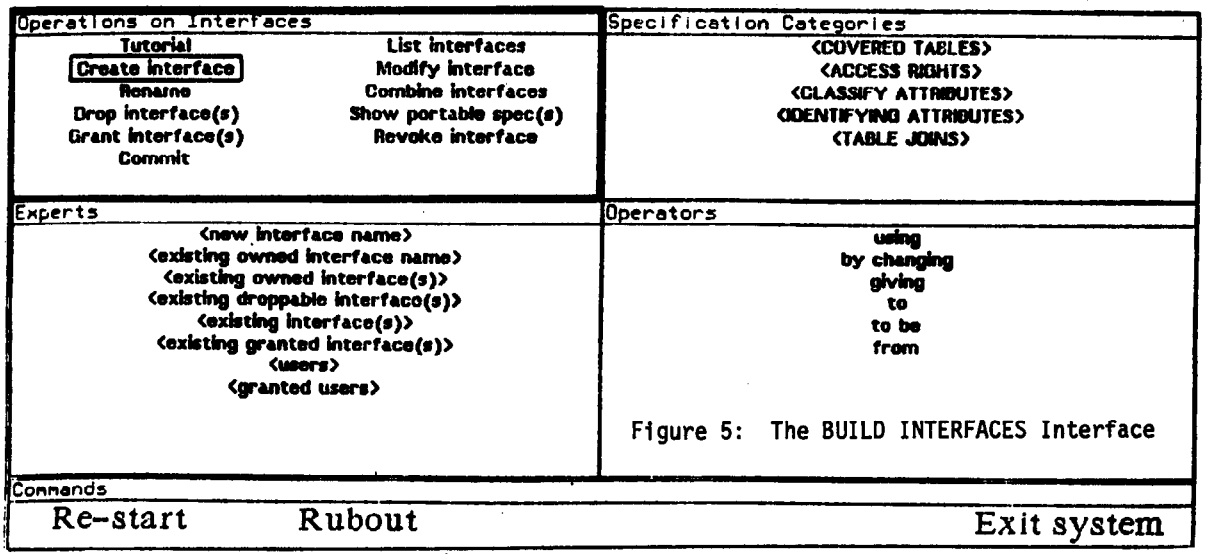


Figure 5: The BUILD INTERFACES Interface