A Natural Language Interface to the NRL-SSC Geospatial Information DataBase*

Kenneth Wauchope and Stephanie S. Everett
Navy Center for Applied Research in Artificial Intelligence (NCARAI)
Naval Research Laboratory, Code 5512
Washington, DC 20375-5337

INTRODUCTION

The Geospatial Information DataBase (GIDB™) developed by the DMAP (Digital Mapping, Charting & Geodesy Analysis Program) group at the Naval Research Laboratory's Stennis (Mississippi) Space Center facility is a Geographic Information System that provides networked, GUI-based query and display of geospatial data accessed from a variety of servers via an object-oriented database approach [McCreedy et al. 2002]. At the time of the work described in the current report (FY2001), GIDB was implemented as a Java applet that could be downloaded in a Web browser from an NRL-SSC server and used to access any of over sixty geospatial databases at that facility via a Java servlet called WMT (World Map Tools). Since GIDB was being integrated into the U.S. Marine Corps' Capable Warrior exercise and our Speech and Natural Language group at NCARAI had recently completed development of a speech interface to Capable Warrior's Battlespace Visualization Tool [Wauchope 2000], we proposed to provide a natural language (NL) interface to GIDB based on our previous experience developing the InterLACE natural language interface to a military cartographic database [Wauchope 1996]. This report describes the technical details of that implementation.

THE GIDB APPLET

Figure 1 shows a screen shot of the GIDB applet displaying a portion of the geospatial database for the MOUT (urban warfare practice area) at the Marine Corps' Camp LeJeune, NC. The GUI has just been used to highlight all the public (i.e. non-commercial) buildings such as schools and churches, and the user is about to submit a followup request to be shown the subset of those buildings that have flat roofs. This example exhibits several of the limitations of using traditional GUIs to perform database queries of this sort. First, it may not be immediately obvious to the user that the *Theme:Feature Class* pair **Population:Buildings Areas** represents non-commercial buildings whereas the distinct **Industry:Building Areas** refers to commercial buildings. Second, two separate queries must be performed to get the final desired result, the first obtaining the set of all public buildings and the second (via the *Feature:Value* pair **Structure Shape of Roof:Flat**) the subset of those with flat roofs. The user is not able to view all buildings both public and private with flat roofs in a single display, since the two Feature Classes (though similarly named) belong to two different Themes and so must be queried and displayed separately. As a final example, to locate all houses of worship the user would query public buildings for the feature-value pair **Building Function Category:House of Worship**, but to locate a church would instead query for the very different pair **House of Worship Type:Church**.

* AIC Technical Report AIC-03-002, Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Code 5512, 4555 Overlook Ave. SW, Washington DC 20375-5337, March 2003.

A natural language interface has two primary advantages over this sort of traditional GUI-based database query methodology. First, the user is not constrained to the possibly arbitrary hierarchy of field names implemented in any particular database, but can use more natural terminology (e.g. *buildings, public buildings, houses of worship, churches*) which the system then automatically maps to the relevant combination of database classes, features and values. Second, natural language syntax allows the nesting of multiple class and/or attribute-value constraints into a single query (*Which public buildings with flat roofs are within 30 meters of the church?*) whereas the GIDB applet requires three separate queries to obtain that result: show all public buildings with flat roofs, show all churches, and then find all results from query 1 that are within 30 meters of the results from query 2.

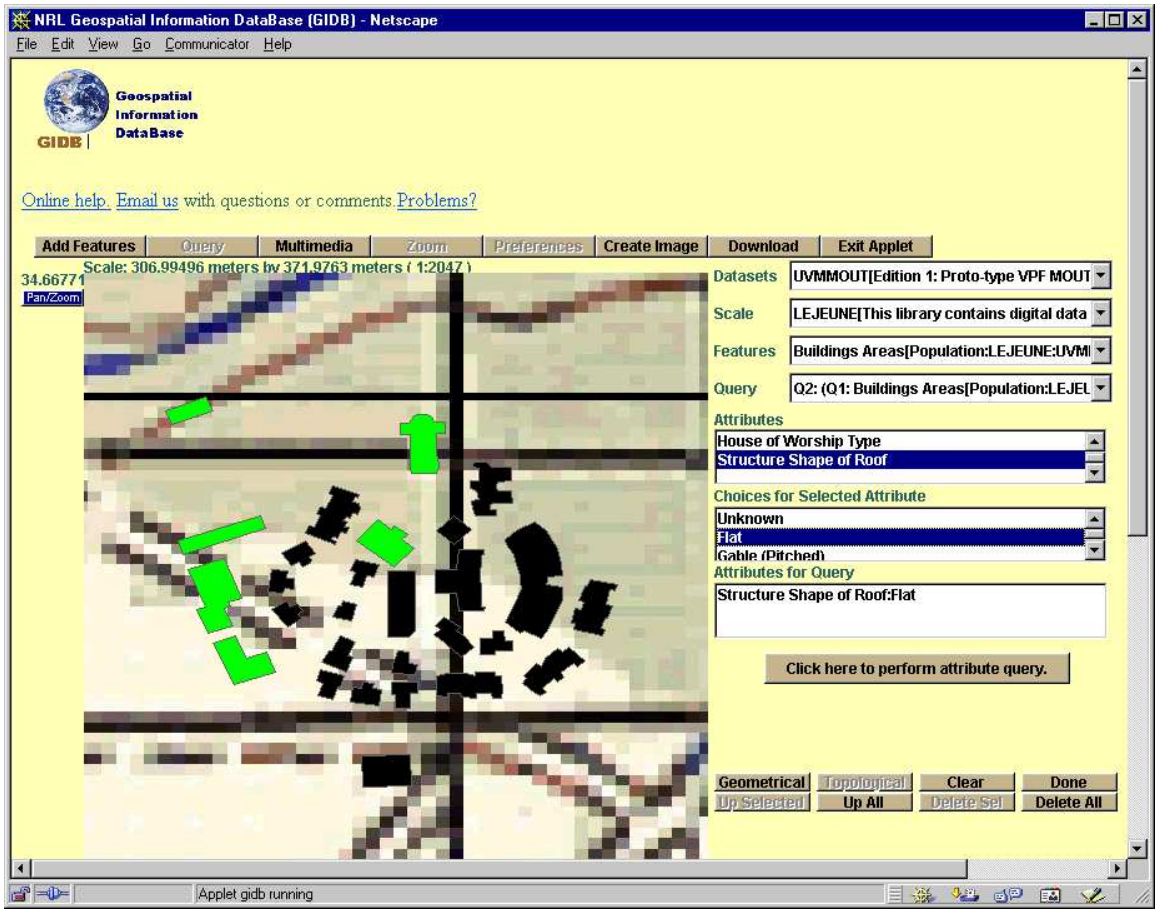

Fig. 1 – GIDB Applet displaying Population: Buildings Areas

SYSTEM ARCHITECTURE

The initial technical problem we encountered in this project was to find a way to link our natural language processor (written in Common LISP, an interpreted programming language) to Java (another interpreted language) so as to remotely connect to the WMT Java servlet for access to the geospatial databases at Stennis. While we had considerable prior experience linking machine-executable object code into LISP using its foreign function interface, Java bytecodes are not directly executable but require the presence of a Java virtual machine to interpret them. An alternative with which we also had considerable experience was to have the NL processor and target application communicate by passing strings over a TCP/IP socket connection, but that

would have required writing a fairly sophisticated command interpreter for invoking Java functions and returning the indices of stored Java objects to serve as the referents of noun phrases and pronouns on the LISP side. Fortunately, Franz Inc. had just the previous year released a new Java linking package for Allegro Common LISP called jLinker, which proved to be exactly what was needed by allowing direct LISP calls to Java functions and LISP pointers to Java objects.

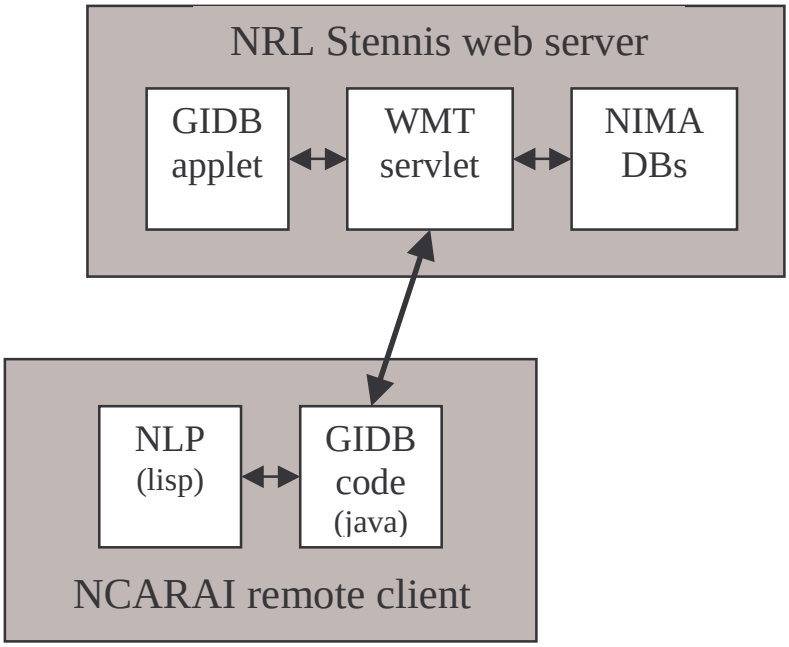

Fig. 2 – GIDB NL Interface System Architecture

Figure 2 shows the resulting system architecture. On the Stennis server, the GIDB applet accesses the geospatial databases via the WMT Java servlet. At NCARAI, the same servlet is accessed remotely over the network by a block of Java code written using GIDB library functions and interfaced to the NL processor using the jLinker package. Thus the natural language interface does not interact with the GIDB applet itself, although implementing such a multimodal interface would have been a desirable followup project.

NATURAL LANGUAGE INTERFACE

The NL interface is built using NAUTILUS [Wauchope 1994], a general-purpose NL processor developed in-house at NCARAI that has been used to interface to a wide variety of DoD-sponsored applications including an air combat command and control simulation, a Virtual Reality tactical air warfare display, two map-based military simulation systems, and NRL's semiautonomous mobile robots. Of the map-based interfaces, the one most relevant to the current project was the InterLACE multimodal interface to a large, real-world cartographic database of (then) East Germany that included such spatial features as towns, roads, lakes, waterways and powerlines. InterLACE provides full NL database query capability that can answer questions such as *What's the nearest small town to Leipzig?* and *Are there any airstrips northwest of there?* Because of its close similarity to a Geographic Information System, a considerable amount of the InterLACE grammar and lexical semantics was thus directly portable to the GIDB application.

For demonstration purposes we chose as our domain the UVMMOUT geospatial database (Marine Corps Camp LeJeune, NC) as being highly representative as well as appropriate for our

sponsors at the Marine Corps Warfighting Laboratory, Quantico, VA. This database represents eight Themes each containing two to four Feature Classes and covering a total of over fifty Features, each taking up to a dozen possible values. The vocabulary for the interface consists of just over 570 words, mapping to approximately 230 lexical semantic classes.

The next step in constructing a NAUTILUS interface is the writing of so-called Translation Functions, which are LISP routines that define the mapping from NAUTILUS logical predicates to application-specific codes called Interface Functions. The GIDB interface required the definition of approximately 90 Translation Functions such as `GET-DISTANCE-FROM` (*How far is the church from City Hall?*), `HAVE-ROOF` (*Does it have a pitched roof?*) and `GET-HEIGHT` (*What's the height of the US Embassy?*), many of which were modeled in large part after InterLACE predecessors.

The final and most labor-intensive step required to interface NAUTILUS to a new application is the coding of the Interface Functions. In this project we first map the Translation Functions to about thirty lower-level LISP procedures that handle the jLinker interactions between LISP and the roughly twenty Interface Functions, which are comprised of approximately 500 lines of Java code. These include such routines as `distBetween` (find the shortest Great Circle distance between two points on the coordinate vectors of two database objects) and `attval` (see if a database object has an attribute of a given name with a given value). In an earlier cut of the system we experimented with writing most of the Interface Function code in LISP jLinker calls and less in pure Java, but found that runtime speed increased substantially the less we utilized the slow jLinker interface.

The following example illustrates the successive steps by which NAUTILUS processes the simple query *How tall is the church?*

1. Regularized parse tree:

```
(ASKWH (WHICH N6 TALL)
       (PRESENT BE (THE N5 CHURCH SINGULAR) VAR))
```

2. Case frame semantic interpretation:

```
(ASKWH (WHICH N1 (:CLASS HEIGHT))
       (PRESENT #:V9155 (:CLASS GET-HEIGHT)
                (:CARRIER(THE N0(:CLASS CHURCH) SINGULAR))
                (:POSSESSED N1)))
```

3. Quantified logical form:

```
(TELL (SETOF X3 (SETOF N1 HEIGHT)
            (EXISTS! X4 (SETOF N0 CHURCH)
                  (GET-HEIGHT :CARRIER X4 :POSSESSED X3))))
```

4. Discourse entity (reference resolution) for the noun phrase *the church*:

```
(N0 SINGULAR CHURCH :CARRIER
    #<TRAN-STRUCT Java IP 1077,121086247
                  WmtModule.VectorFeature>)
```

Note in step 4 that the noun phrase's referent (the final element in the discourse entity list) is the jLinker representation of the actual Java object downloaded from the Stennis Java servlet. The following is an abbreviated LISP trace of the steps involved in that dereferencing process.

```
1:   (TINSEL-WMT-DEREFERENCE CHURCH NIL)
      2: (WMT-DOWNLOAD "Population" "Buildings Areas")
      2: returned #<TRAN-STRUCT Java IA1 1043,121086247,109
                                    [LwmtModule.VectorFeature;>
      2: (WMT-FIND-MATCHING-VALS
                 #<TRAN-STRUCT Java IA1 1043,121086247,109
                                    [LwmtModule.VectorFeature;>
                  "House of Worship Type" "Church")
      2: returned (#<TRAN-STRUCT Java IP 1077,121086247
                                     WmtModule.VectorFeature>)
```

As the trace shows, NAUTILUS first determines that semantic class `CHURCH` corresponds to an object with Theme `"Population"` and Feature Class `"Buildings Areas"`. Since that particular block of objects has not yet been downloaded from the database server, NAUTILUS does so and caches the result locally to be used in resolving subsequent references to other public buildings. It then searches the resulting vector of objects for one or more with a `"House of Worship Type"` feature having value `"Church"`, and returns the result. Since there is only one such matching object, the singular definite NP *the church* succeeds without raising a number error, allowing processing to continue.

The next step is the evaluation of the logical form in step 3, which roughly translated says "Tell me all the pairings [`SETOF`] of the attribute `HEIGHT` with the height value [function `GET-HEIGHT`] of the unique object `CHURCH`." First, the function `GET-HEIGHT` is evaluated in the context of the logical form. `GET-HEIGHT` is defined by the following two LISP Translation Functions:

```
(defun GET-HEIGHT (&key carrier possessed)
  (let ((val (HEIGHT-OF carrier possessed)))
    (if (numberp val)(format nil "~A meters" val) val)))

(defun HEIGHT-OF (carrier possessed)
  (when (consp possessed)(setq possessed (first possessed)))
  (let* ((val (case possessed
               (predominant-height
                (attrib carrier "Predominant Height"))
               (height-above-surface-level
                (attrib carrier "Height Above Surface Level"))
               (height
                (or (attrib? carrier "Predominant Height")
                  (attrib? carrier "Height Above Surface Level")
             ;; Armistice Line "Name 4" attribute = "HEIGHT = 1.6"
                  (attrib? carrier "Name 4")
                  (undefined-attribute "Height" carrier))))))
    (if (stringp val)(read-from-string val) val)))
```

In the context of our current example, the Translation Function call is

```
(GET-HEIGHT
  (#<TRAN-STRUCT Java IP 1077,121086247 WmtModule.VectorFeature>)
  'HEIGHT)
```

which in turn invokes function HEIGHT-OF to get the numeric value of the object's height. Since *height* can refer to two possible attributes in this particular database (the predominant height of a grove of trees, or the height above ground level of a building, landmark or obstruction), HEIGHT-OF makes the call

```
(attrib?
  (#<TRAN-STRUCT Java IP 1077,121086247 WmtModule.VectorFeature>)
  "Height Above Surface Level")
```

only after first determining that the church object does not possess a "Predominant Height" attribute. attrib? is a low-level routine that simply tests a database object for an attribute of a given name, returning the attribute's value if present and NIL otherwise. It is defined as follows:

```
(defun attrib? (obj name)
  (and (typep obj 'TRAN-STRUCT)
       (let ((result (jcall (jmeth "WMTintfuns" "attrib" 2)
                            NLI obj name)))
         (if (equal result "NIL")
             nil
           result))))
```

attrib? uses the jLinker interface to call the Interface Function attrib, a Java method in the WMTintfuns class and defined as follows:

```
public java.lang.Object attrib(WmtModule.VectorFeature obj,
                          java.lang.String name) {
      WmtModule.Attribute[] atts = obj.attributes;
      for (int i = 0; i < atts.length; i++) {
          WmtModule.Attribute att = atts[i];
          if (att.name.equals(name)) return(att.value);
      }
      return "NIL";
    }
```

From this point on, all Java methods are imported from classes provided by the GIDB servlet developers at Stennis. Having obtained a "Height Above Surface Level" value of 30, attrib passes this value back to the Translation Function GET-HEIGHT which formats it as the string "30 meters". Returning to the evaluation of the Logical Form

```
(TELL (SETOF X3 (SETOF N1 HEIGHT)
             (EXISTS! X4 (SETOF N0 CHURCH)
                      (GET-HEIGHT :CARRIER X4 :POSSESSED X3))))
```

the outermost SETOF operator now pairs up the attribute HEIGHT with this value to create the list structure (HEIGHT "30 meters"). Finally, the TELL performative is responsible for reporting this value back to the user in an English-like syntax. Since NAUTILUS was designed primarily as an interpreter of natural language input, its built-in NL output facilities are deliberately quite spare – the placeholder definition of TELL currently just generates the sentence fragment "Height: 30 meters" as its reply. While replies of this sort are concise and "natural" in the sense of not being annoyingly verbose, there are two advantages to having the system instead respond in complete English sentences. First, it reassures users that their queries have been correctly understood and interpreted, and second, it "primes" users to avoid using sentence fragments in their own queries since such inputs are more difficult for the NL processor to

interpret unambiguously. In the InterLACE project we used a full-fledged semantically-based NL generation system FUF/SURGE [Eldahad 1992, Robin 1994] to provide the option of full-sentence query responses. In the current project we instead opted to implement a simpler, template-based syntactic approach to NL generation, described next.

NATURAL LANGUAGE GENERATION COMPONENT

The NL generation component accesses the parse tree of the user's input query to formulate a full-sentence response that echoes the user's vocabulary and phrase structure. For example, the system responds to the query *How tall is the church?* with "The church is 30 meters tall", but answers the query *What is the height of the church?* with "The height of the church is 30 meters". Figure 3 shows some example input and response patterns. Angle-bracketed symbols represent the words subsumed by particular nodes in the query parse. For example <S-TPOS> represents the contents of the S (SUBJECT) node minus the contents of its TPOS (determiner) child node, <NEG> represents an optional *not*, and <O> represents the OBJECT node (verb complement). Thus for the query *Are any warehouses within 300 meters of the church?* the determiner *any* is stripped from the sentential subject *any warehouses* (and the noun *warehouse* singularized) to generate the response "Yes, **one warehouse is** within 300 meters of the church". If the reply is negative, its form depends on whether the input query was itself negative: *Are any warehouses within 30 meters of the church?* generates "No, none of the warehouses is within 30 meters of the church", but the query *Are any warehouses not within 300 meters of the church?* generates "No, all warehouses are within 300 meters of the church".

Query Pattern	Response Pattern
Are any (of the) <S> <NEG> <O>	"Yes, [#] <S-TPOS> is/are <NEG> <O>"
Are any (of the) <S> <O>	"No, none of the <S-TPOS> is <O>"
Are any (of the) <S> not <O>	"No, all <S-TPOS> are <O>"

Fig. 3 – Sample Query and Response Patterns for Natural Language Generator

CONCLUSIONS

This project demonstrates the feasibility of interacting with a Java servlet-based Geospatial Information System back end via a natural language front end. Natural language and graphical interfaces offer complementary ways of accessing this information: the graphical interface is particularly useful when the objects under consideration are physically adjacent and fully visible in the display, while the NL interface is more powerful at accessing sets of objects based on symbolic descriptions of their types and attributes. We believe the ideal interface to such a system would be multimodal, with both access options available simultaneously and mutually integrated.

REFERENCES

M. Eldahad [1992]. Using Argumentation to Control Lexical Choice: A Functional Unification Implementation. Ph.D. thesis, Computer Science Department, Columbia University.

F. McCreedy, R. Ladner, R. Wilson, J. Breckenridge, S. Carter, H. Mesick, D. Olivier, K. Shaw, C. Baribault, T. Lovitt [2002]. System Documentation for the Geospatial Information DataBase (GIDB™) System Server Version 2.0, NRL/MR/7440--02-8275, June 24, 2002.

J. Robin [1994]. Revision-Based Generation of Natural Language Summaries Providing Historical Background. Ph.D. thesis, Computer Science Department, Columbia University.

K. Wauchope [1994]. "Eucalyptus: Integrating Natural Language Input with a Graphical User Interface," NRL Technical Report NRL/FR/5510--94-9711, Washington, DC: Naval Research Laboratory, 1994. http://www.aic.nrl.navy.mil/papers/1994/AIC-94-007.ps

K. Wauchope [1996]. Multimodal interaction with a map-based simulation system. NCARAI Technical Report AIC-96-027, Naval Research Laboratory, Washington DC, July 1996. http://www.aic.nrl.navy.mil/papers/1996/AIC-96-027.pdf

K. Wauchope [2000]. Voice control for the Battlespace Visualization Tool. NCARAI Technical Report AIC-00-005, Naval Research Laboratory, Washington DC, April 2000. http://www.aic.nrl.navy.mil/papers/2000/AIC-00-005.pdf