LUKE: AN EXPERIMENT IN THE EARLY INTEGRATION OF NATURAL LANGUAGE PROCESSING

David A. Wroblewski, Elaine A. Rich
MCC
Human Interface Laboratory
3500 West Balcones Center Drive
Austin, Texas 78759

Abstract

Luke is a knowledge editor designed to support two tasks; the first is editing the classes and relations in a knowledge base. The second is editing and maintaining the semantic mapping knowledge neccesary to allow a natural language interface to understand sentences with respect to that knowledge base. In order to emphasize design decisions shared between the two tasks, Luke provides facilities to concurrently debug the application and the natural language interface. Luke also makes natural language available in its own user interface. This makes it possible for a knowledge base builder to exploit natural language both as a way of locating desired concepts within the knowledge base and as a a way of doing consistency checking on the knowledge base as it is being built.

Introduction

Luke is a knowledge base editor that has been enhanced to support entering and maintaining the semantic mappings needed by a natural language interface to a knowledge base. Thus Luke supports a team of system builders who are simultaneously building a knowledge-based program and building a natural language interface to that program. It makes sense for a single tool to support both of these efforts because the efforts themselves are logically intertwined in two important ways, both of which result from the fact that the application program and its NL interface must share a single knowledge base. (This sharing is necessary because otherwise the NL system will not be able to communicate with the application). The first way in which the two efforts Luke supports are related is that, although they produce two systems that are different and may thus place different demands on their associated knowledge bases, both must share a single such knowledge base. By supporting the early integration of the application program and the NL interface as this single knowledge base is being built, Luke helps to ensure that it will be adequate, with respect to both its content and its structure, to support both these target tasks. The second way in which the two system building tasks are related is that one can support the other. By associating natural language with concepts as they are entered into a knowledge base, Luke makes natural language available in its own interface. This makes it possible for the knowledge base builder to exploit natural language both as a way of referring to objects in the knowledge base and as a way of doing consistency checking on the objects themselves. In this paper, we will describe both what Luke does and how doing that supports this productive view of the interaction between building a knowledge based system and building an associated natural language interface.

Background And Motivation

A Model Of Semantic Analysis

All of the following discussion is based on a model of semantic analysis similar to that proposed in (Hobbs, 1985). Under this model, syntactic and semantic analysis are done as separate operations. The first stage of semantic analysis is a conversion to *initial logical form*, in which the surface content of the sentence is encoded in a set of expressions that look like logical terms, but whose predicates are taken directly from the words used in the sentence. Initial logical form captures the predicational structure of the sentence, without expressing it in terms of the knowledge base.

Once produced, the expressions in initial logical form are individually translated into *final logical form*, which is a set of first-order terms whose predicates are those used in the application's knowledge base. The translation from initial logical form to final logical form is done via a set of rules known as *semantic mappings*, and it is the acquisition of these semantic mappings that is the subject of this paper[1]. The control of and exact details of semantic mappings are irrelevant for this

[1]In reality, we further subdivide the semantic mappings into *mappings* and *compoundings*. Mappings we described above. Compoundings are rules that specify how two nouns can be compounded.

discussion; it is enough to know that semantic mappings roughly translate from the surface form of the English input to expressions built in terms of the target knowledge base.

The general form of a semantic mapping is shown below, along with several examples. A semantic mapping is a rule for translating one initial logical form into zero or more final logical forms. A *semantic lexicon* is then a collection of semantic mappings that specify translations for the words in the syntactic lexicon.

Generally:
ilf --> flf_1, flf_2, ..., flf_n

Examples:

```
(dog ?x) --> (canine ?x)             (1)

(make ?i ?x ?y) -->                  (2)
                  (creating ?i)
                  (agent ?i ?x)
                  (object ?i ?y)
                  (graphic-obj ?y)
```

A mapping for the noun "dog" is shown in (1). This rule states that the referent of a noun phrase whose head is "dog" must be a member of the class `canine`. Mapping (2) shows that sortal restrictions can be included in the mapping, in this case restricting the direct object of the verb "make" to be a member of the class `graphic-obj`. An ILF may match the left hand side of many semantic mappings, and so ambiguity is captured in the semantic lexicon.

In our model of semantic analysis, these semantic mappings are used to build a picture of what was said in the sentence by posting constraints. In fact, each semantic mapping exploits two kinds of constraints. Lexical constraints define the applicability of a mapping as a function of the words that appear in a sentence. These constraints always appear on the left hand side of a semantic mapping. Knowledge-base constraints define the applicability of a mapping as a function of the meanings of the current word, as well as the other words in a sentence. These constraints always appear on the right hand side of a semantic mapping. Viewed this way, mapping (1) constrains the referent of "a dog" (or "the dog" or any noun phrase with "dog" as its head) to be a member of the class `canine`, but does not specify what (if any) specialization of `canine` the referent might refer to. For example, it does not commit to the class `schnauzer` versus the class `dachshund`.

Past Experience

Our early attempts at porting our natural language understanding system, Lucy (Rich, 1987), consisted of "hand-crafting" a set of semantic mappings for an existing knowledge base. The application program was an intelligent advice system (Miller, 1987) that would accept questions from a user about operating a statistical analysis program and try to provide advice based on its knowledge of the program's interface and internal structure.

Creating the semantic mappings was a long and tedious chore. Starting with a mostly-complete knowledge base, finding the correct semantic mappings was a matter of knowledge acquisition, in which we asked the knowledge base designers what knowledge structure a particular word might map onto. Many times this was almost as difficult for the knowledge base designers as it was for the "semanticians", since the knowledge base was quite large, and developed by several people. Often, the knowledge base designer being interviewed was not familiar with the area of the knowledge base being mapped into, and thus could not accurately answer questions, especially with respect to completeness (i.e., "this is the *only* class that the word could map into.")

Furthermore, defining the semantic mappings often uncovered inconsistencies in the knowledge base. When this happened, it was not always immediately clear what the correct action was; we could either fix the knowledge base or live with the inconsistencies (which usually meant semantic ambiguity where none was really necessary.)

Even worse, there were many cases where defining *any* semantic mapping was problematic. In these cases, representational decisions that had already been made either precluded or made very difficult any principled mapping of English expressions into the knowledge base. This happened when information was needed to analyze a syntactic constituent (perhaps a noun phrase like "the mouse") but the referent of the constituent (the mouse icon on the screen), was not represented in the knowledge base. Thus, no semantic mapping could be written. The problem could be solved by simply introducing the relevent knowledge, but sometimes a better solution would have involved redesigning a portion of the knowledge base to represent more clearly important features of the domain. Usually this was too costly an option to consider.

Finally, we quickly discovered that the dream of establishing the semantic mappings once and for all was a fallacy. Any significant knowledge base is "under construction" for a long period of time; introducing semantic mappings before the knowledge base is totally done necessarily implies maintenance of the semantic mappings in the face of a changing knowledge base. This is a paradox: on the one hand, it would be best to have a completed knowledge base before doing any semantic mapping. On the other hand, to avoid problematic semantic mappings it would be best to introduce semantic mappings and "debug" them as early as possible in the development of the knowledge base.

The Dual-Application Development Model

In order to avoid the problems mentioned in the last section, Luke endorses a *dual-application model* of the development process. Under such a model, there are two related applications being developed. One is a *natural language interface* (NLI), responsible for forming a syntactic, semantic, and pragmatic analysis of user input, and passing the interpreted input to the *domain application.* The domain application, of course, could be anything. We focused on knowledge-based applications so that we could assume that a knowledge base was a part of the domain application. We assume that the natural language understanding component and the domain component both have access to the knowledge base, and that semantic analysis should be done with respect to that knowledge base.

The dual-application model highlights the design interplay between the domain application and the interface. In particular, knowledge base design decisions motivated exclusively by the domain application or the NLI, without regard for the other application, are likely to be inadequate in the final, integrated, system. Such ill-informed decisions might be avoided in a development environment that allows the earliest possible integration of the applications. Luke is our first attempt to provide such an environment, and is built to support the work done during early prototyping and full-scale development of an application.

Luke's Architecture

Luke is an enhanced version of a simple knowledge editor, as illustrated in Figure 1. In the discussion that follows, we will refer to this as the *base editor*, because it forms the foundation upon which Luke is built. All operations performed at the editor interface are translated into a series of function calls via a well-defined functional interface to the knowledge representation system. The base editor is a complete system: it can be run independently of any of the extensions described hereafter. The base editor knows nothing of the Lucy natural language understanding system.

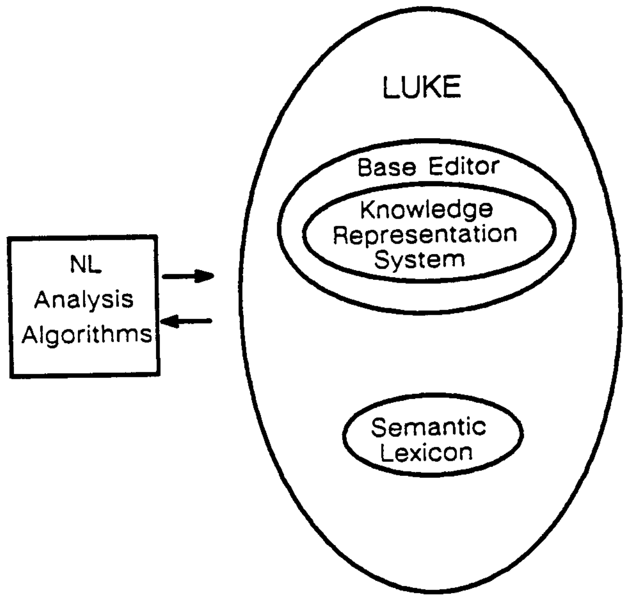


Figure 1: Luke's Architecture

The base editor allows two types of commands: *terminological* and *assertional* commands[2]. These terms are taken from (Brachman, 1983), which defines a knowledge base as consisting of two "boxes". The *Tbox* holds the terminological information of the knowledge base, information that defines what symbols are valid class identifiers, and what the names, arities, domains and ranges of those relations are. Brachman and Levesque liken the terminological knowledge to the "noun phrases" of the knowledge base.

[2]Actually, there is at least one other type of command: *management*. Management commands handle such prosaic issues as saving and loading knowledge bases. While these commands will not be described in detail in this paper, the reader should be aware that a significant effort was also required to upgrade these to handle managing both the knowledge base and the semantic lexicon.

Table 1: Knowledge Editing Operations and Their Effects

Operation	Semantic Lexicon Effect

`Create Class`
`Create Slot`
	New mappings possible.
	Old mappings may have to be refined.

`Delete Class`	Existing mappings may be invalid because they refer to a now nonexistent class.
`Delete Slot`	Some existing mappings may be invalid because they refer to a now nonexistent slot.

`Attach Superclass`
`Detach Superclass`
	Some existing mappings may be invalid because inheritance paths have changed.

`Rename (anything)`	Existing mappings may be invalid due to renaming.

The *Abox* holds assertional information, described by using logical connectives such as "and", "or" and "not" and the predicates defined in the Tbox to form logical sentences. While the terminological component describes what it is possible to say, the assertional component holds a *theory of the world*: a set of axioms describing the valid inferences in the knowledge base.

As shown in Figure 1, Luke extends the base editor by additionally maintaining a semantic lexicon. Each time an operation is performed on the knowledge base, Luke must update the semantic lexicon so that the set of semantic mappings it contains remains consistent with the updated knowledge base. Table 1 shows some operations and their effect on the semantic lexicon.

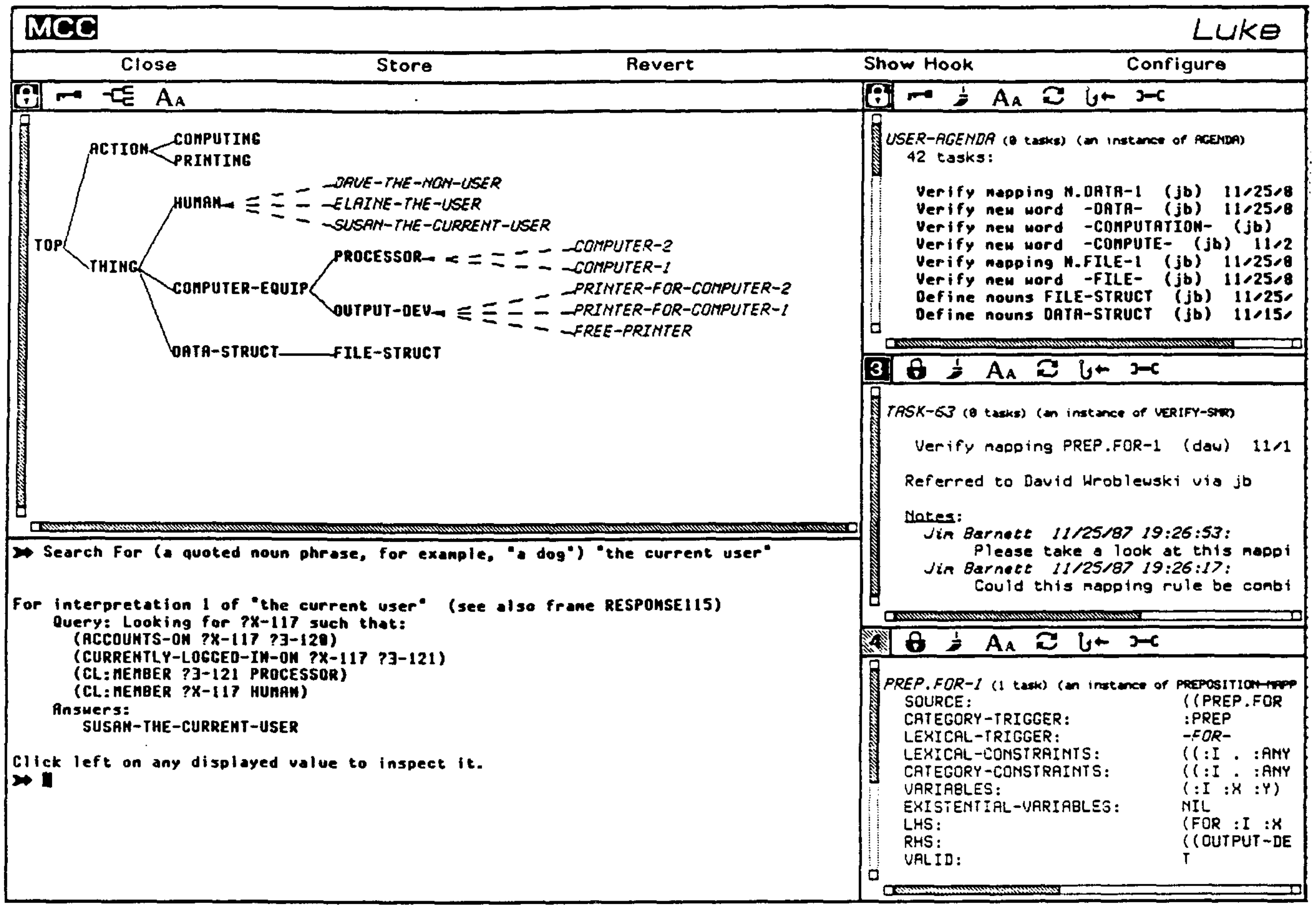
As can be seen from this table, operations that change the terminological content of the knowledge base (such as `Create Class` or `Create Slot`) may change the number or structure of the semantic mappings known. For example, consider the case of the `Create Class` command. By adding a new class to the knowledge base, we have extended the *Tbox*; since the knowledge base is now able to describe something it could not describe before, some English noun phrases that were previously uninterpretable can now be mapped into this class. Existing mappings may have to be changed, since the act of adding a class may constitute a refinement of an existing class and its associated mappings.

For instance, one might add a set of subclasses of `canine` where none used to exist. If the current set of semantic mappings map "poodle" and "doberman" into `canine`, then these rules may have to be refined to map into the correct subclass. Extending the terminological component of the knowledge base extends the range of or precision with which syntactic constituents may be semantically analyzed.

Operations that alter the *Abox* have less well-defined effects on the semantic lexicon. For instance, without detailed knowledge of the domain application and the domain itself, the addition of an inference rule to the knowledge base implies nothing about the possible semantic mappings or the validity of current mappings. In general, it is very difficult to use the assertional component of a knowledge base during semantic processing; for this reason, we will concentrate on terminological operations for the remainder of this paper.

Luke, then, is a "base editor" extended to account for the semantic mapping side effects of knowledge editing operations. Luke reacts in predictable ways to each editing operation, based on the information shown in Table 1:

- **New mappings possible**: Luke reacts to this condition by conducting an "interview" with the user. Each interview is designed to collect the minimum information necessary to infer any new semantic mappings. In a word, the response to possible new mappings is "acquisition".
- **Old mappings possibly invalid**: Luke reacts to this condition by trying to identify the affected mappings and requesting the user verify their correctness. In a word, the response to possibly invalid mappings is "verification".



```
≫ Search For (a quoted noun phrase, for example, "a dog") "the current user"


For interpretation 1 of "the current user"  (see also frame RESPONSE115)
     Query: Looking for ?X-117 such that:
       (ACCOUNTS-ON ?X-117 ?3-120)
       (CURRENTLY-LOGGED-IN-ON ?X-117 ?3-121)
       (CL:MEMBER ?3-121 PROCESSOR)
       (CL:MEMBER ?X-117 HUMAN)
     Answers:
        SUSAN-THE-CURRENT-USER

Click left on any displayed value to inspect it.
≫ ▮
```

```
USER-AGENDA (0 tasks) (an instance of AGENDA)
   42 tasks:

    Verify mapping N.DATA-1  (jb)  11/25/8
    Verify new word  -DATA-  (jb)  11/25/8
    Verify new word  -COMPUTATION-  (jb)
    Verify new word  -COMPUTE-  (jb)  11/2
    Verify mapping N.FILE-1  (jb)  11/25/8
    Verify new word  -FILE-  (jb)  11/25/8
    Define nouns FILE-STRUCT  (jb)  11/25/
    Define nouns DATA-STRUCT  (jb)  11/15/
```

```
TASK-63 (0 tasks) (an instance of VERIFY-SMR)

   Verify mapping PREP.FOR-1  (daw)  11/1

  Referred to David Wroblewski via jb

  Notes:
    Jim Barnett  11/25/87 19:26:53:
         Please take a look at this mappi
    Jim Barnett  11/25/87 19:26:17:
         Could this mapping rule be combi
```

```
PREP.FOR-1 (1 task) (an instance of PREPOSITION-MAPP
  SOURCE:                           ((PREP.FOR
  CATEGORY-TRIGGER:                 :PREP
  LEXICAL-TRIGGER:                  -FOR-
  LEXICAL-CONSTRAINTS:              ((:I . :ANY
  CATEGORY-CONSTRAINTS:             ((:I . :ANY
  VARIABLES:                        (:I :X :Y)
  EXISTENTIAL-VARIABLES:            NIL
  LHS:                              (FOR :I :X
  RHS:                              ((OUTPUT-DE
  VALID:                            T
```

Figure 2: The Luke Window

Base Editor Facilities: Windows and Agendas

Figure 2 shows the screen as it might typically appear during an editing session with Luke. The user is provided with a suite of inspectors to display the class hierarchy or view individual frames in detail. Each inspector provides an iconic menu of operations that can be performed on it or its contents. Components of frames in the inspectors, such as the names of slots, are mouse-sensitive and provide the main machanism for editing the frames themselves.

Also provided is an agenda of tasks to be performed. A user may manually queue up tasks to perform as reminders, annotate tasks, or refer tasks to other members of the development team. Tasks may be scheduled automatically as a side effect of various editing commands. There are two main types of tasks: *verification tasks* and *acquisition tasks.* Verification tasks are reminders to inspect some part of the knowledge base to ensure its consistency. Acquisition tasks are (typically) interviews that Luke has requested with the user.

The base editor also provides a method of *delaying* tasks. Some tasks, such as acquisition tasks, are started at a default time, usually immediately after the action that inspired them. The user has the option, at any point during the task, of pressing the *delay key*, causing the task to be stopped, and an agenda item created for it if none already exists. Through this delaying mechanism, the user has control of when tasks are executed.

The agenda is shown in the upper right inspector in Figure 2. It is implemented as a frame (an instance of the built-in class agenda, and may be inspected via the normal editing commands of the base editor. Each task is represented as an instance of the class task, and includes a description of the event that inspired it. Although the base editor makes very little use of the agenda mechanism, Luke schedules a large number of interviews and verification tasks through the agenda.

User Tasks, User Models

Luke is different from most other tools of its kind for three reasons. It provides support for both the acquisition and maintenance of semantic mappings. Because it then knows those semantic mappings, it makes natural language available in its own interface. And in order to do these things, it must assume more sophistication on the part of its users. The intended users of Luke are members of a knowledge engineering team. These people are assumed to be familiar with the content and structure of the knowledge base, or to be capable of discovering what they need to know by inspecting the knowledge base. Although they are not assumed to have an extensive linguistics background nor extensive familiarity with the implementation of the semantic processing algorithms of Lucy, they are assumed to have a "qualitative model" of semantic processing (as presented earlier). Moreover, since we assume that a *team* of engineers will be building the applications, some with special interests or talents, tasks that might require greater linguistic sophistication may be delayed until the "linguistics specialist" can be brought in.

Luke provides tools for the acquisition of semantic mappings and the maintenance of those mappings once collected. Although traditionally, little attention has been paid to the latter task, we believe that it may prove to be the more important of the two; once a large base of mappings has been established, it is only practical to maintain them with tools specifically designed for that task. The next part of of this section will describe tools provided by Luke for both tasks. Then the remainder of the section will show how these mappings can be used to inhance the user interface of Luke itself.

Acquiring Semantic Mappings

The Luke acquisition modules are built with the following design guidelines:

1. Perform acquisition tasks temporally near the event that causes them.
2. Allow the user to delay acquisition at will.
3. Allow the user to specify the minimum information from which semantic mappings can be deduced.
4. Remember that people are better at verifying a proposed structure than they are at creating correct structures from scratch.
5. Try to repay the user for the work expended in the interviews by using the semantic mappings for knowledge base debugging, navigation, and consistency checking.
6. Project a correct model of semantic processing to the user throughout the acquisition process.

In the Luke environment, acquiring semantic mappings turns out to be quite simple. The scheme we use in Luke involves a three-stage process. In the first stage, Luke collects *associations*. Simply put, an association is a triple of the form

```
<word,part-of-speech,structure>
```

In the second stage, a set of heuristics inspects the associations and compiles them into semantic mappings. For instance, the association `<"dog",noun,canine>` might be built during acquisition to indicate that some noun sense of the word "dog" maps into the class `canine`. In the final stage, the mapping rule deduced from the association is built, presented to the user for refinement via a special mapping editor, and entered into the semantic lexicon. Occassionally, Luke uses the new mapping to inspire other mappings, such as the nominalizations of a verb. In this case, once a verb mapping is known, nomimalizations of it are collected and created in the same manner, and heuristics take advantage of the fact that the new nouns are nomimalizations of a verb whose mapping is known. Thus the constraints on the complements of the verb are used to generate mappings for prepositions that can be used to specify the complements of the nominalization of that verb.

Although the basic acquisition technique is simple, obeying guideline 6 can be tricky. For instance, in an early version of Luke we temporally separated the interviews from the heuristic construction of associations. Further, we did not submit the mappings to the user when they were guessed. The mappings were guessed later, in a background process, usually invisible to the Luke user. Yet semantic analyses often succeeded, giving users the impression that the associations were driving the semantic analysis routines, not the semantic mappings deduced from them. With such a model of the process, the user was confused and unprepared when semantic mappings ("where did *they* come from?") were incorrect and had to be inspected, debugged, and edited. In the current version of Luke, the semantic mappings are presented to the user at the end of the interview, to be reviewed and edited immediately. Connecting the process of associating with the mapping creation process leads to much less confusion.

Managing the Semantic Lexicon

Once a semantic lexicon exists, maintaining it becomes a significant chore. During routine knowledge base editing a user may change the terminological content in such a way that existing semantic mappings become invalid. Deleting a class, for example, clearly makes any semantic mappings that mention it incorrect. If a large semantic lexicon exists, changing the terminological content of the knowledge base may entail editing a very large number of semantic mappings.

Luke provides a number of tools to help manage the semantic lexicon. These tools fall roughly into two categories, those that support editing and those that aid in consistency checking. The editing tools allow a user to request all the mappings that target a specific frame, or all the mappings that map from a given surface form, via a special mappings browser. Users may edit semantic mappings at any time using the ordinary editing tools of the base editor, because semantic mappings themselves are stored as frames in the knowledge base.

The biggest maintenance service Luke provides is consistency checking. When a frame is deleted, entered, or specialized in the knowledge base, or after any terminological editing operation, Luke collects all of the semantic mappings that might be affected and creates a set of tasks to verify their continuing correctness. As always, the user can choose to handle such tasks immediately, or delay for later consideration.

Exploiting Natural Language in Luke Itself

The overall goal in building Luke is to provide a set of "power tools" (Sheils, 1983) that support the dual application model, and Luke is our first step in that direction. One potential problem in Luke's design is increasing the overhead of building a knowledge base, since various tasks are continually scheduled for the user. This fear is mitigated by the following observations. First, the added overhead doesn't represent extra work to be done by the user, only a different time for the user to do it. If there is to be a NLI for the application, then the developer is in a "pay me now or pay me later" bind, where late payment can be very costly. Viewed this way, Luke is helping the user trade a short-term loss (interviews and verification tasks during editing) for a long-term gain (smaller NLI development effort after the domain application is finished). Second, with the additional information provided by concurrently developing the NLI and the domain knowledge base, Luke can "pay back" the user at editing time by strategically using this information to support both extending and debugging a knowledge base. In the rest of this section we describe some of the ways in which this is done.

Luke provides the `Search For` command, which accepts a noun phrase as its argument. `Search For` converts that noun phrase into a knowledge base query by using the Lucy natural language understanding system. The noun phrase is parsed and semantically analyzed using any known semantic mappings. When the resulting query is executed, the matching frames are stored into a response frame, along with information concerning what mappings were used in the interpretation process. Then the user is presented the frames in a menu. Thus, `Search For` provides both a way of exercising the semantic mappings and retrieving frames from the knowledge base during normal editing. Note that such "retrieval by description" facilities are not usually provided in knowledge editors because it would require a sophisticated query language allowing abstraction and arbitrary user extensions. Because Luke already has access to a natural language analysis component, providing this service to the user is straightforward. Also note that such a service is vital to editing and maintaining large knowledge bases -- finding a frame using just graphical displays of the class hierarchy and detailed single-frame displays does not provide any sort of "random access" capabilities, and finding a specific frame using only such tools can be very difficult.

Luke also provides a method of testing the analysis of entire sentences. The developer can submit a sentence for analysis to the NLI processing algorithms. The analysis of the sentence is returned as a frame in the knowledge base, recording the interpretations found, and a record of the mappings used to get the interpretations. This can be further processed by a "default command loop" used to simulate the behaviour of the application program. Using this facility, it is easy for the application developer to place her/himself in the place of the application program, and to envision the sorts of responses neccesary.

Furthermore, the process of interviewing is a form of documentation. During an editing session, the user leaves throughout the knowledge base a "trail" of semantic hints that various customized commands can take advantage of. For instance, the `Show Associated Nouns` command pops up a quick menu of words associated with the frame in question, providing a handy documentation function.

Finally, Luke can catch several knowledge editing mistakes that the base editor cannot. One of the most common is *class duplication* -- unwittingly creating a class intended to represent the same set of entities as an already-existing class. Often this happens when the knowledge base is being built by a team of people or because it has grown too complex for an individual to visualize. Luke helps solve the problem using the existing semantic mappings. After associating a noun with a class, Luke warns the user of the total number of mappings for that noun and some indication of the frames it might map into. This simple mechanisms detects many cases of class duplication.

Comparison To Other Work

Luke appears to be different than previous systems of its ilk in a number of ways. Most importantly, Luke is built to support the dual-application model of development. Systems such as TEAM (Grosz, 1987), TELI (Ballard, 1986), and to a lesser degree, IRACQ (Ayuso, 1987), all aim for portability between existing, untouchable, applications (usually DBMS's). These tools have generally emphasized building a *database schema* in order to supply the (missing) terminological component of the database. We have rejected such an approach on the grounds that it is only useful for building sentence-to-single-command translators, not for wholesale integration of a NLI with an application. Luke is an attempt to help *design in* the natural language interface from the start.

Because of this basic assumption, Luke is more oriented toward users as sophisticated system builders than as linguistically naive end-users or "database experts". Luke users *will* understand some linguistics, either by educational background, hands-on experience, or special primers and training.

Finally, Luke is designed to support a team of users, not a single user. Luke provides a flexible agenda and task management system that allows users to handle tasks for reviewing existing mappings, investigating potential conflicts in the semantic lexicon, and creating new mappings for new objects in the knowledge base. Such tasks can be operated on in a variety of ways, including scheduling, executing, annotating, or referring them between members of the development team.

Future Plans

At present, Luke is a useful, competent knowledge editor and provides a substrate of tools for concurrently managing the development of an application knowledge base and the NLI that will ultimately operate with it. Ultimately, we hope to make Luke itself a knowledge-based program, adding to it the heuristics that an "expert NLI engineer" might have, and expanding its role to that of an intelligent assistant. The groundwork is laid for such a step; Luke is already driven by a model of itself, the knowledge base, Lucy's algorithms, and its users. In the near term we plan to expand and refine the role that such knowledge plays in Luke's operation.

Acknowledgments

The ideas in this paper are the product of the entire LINGO team. Mike Barnett designed and implemented the agenda facility described herein, and Kevin Knight designed some of the semantic debugging aids. Additionally, most of the ideas about the way this version Luke operates sprang from a working group including Mike Barnett, Jim Barnett, Kevin Knight, and the authors.

References

Damaris M. Ayuso, Varda Shaked and Ralph M. Weischedel. (July 1987). An Environment For Acquiring Semantic Information. *Proceedings of the 25th Annual Meeting of the Association of Computational Linguistics.* .

B.W. Ballard and D.E. Stumberger. (1986). Semantic Acquisition in TELI: A Transportable, User-Customized Natural Language Processor. *Proceedings of the 24th Annual Meeting of the Association of Computational Linguistics.* .

R.J. Brachman, R.E. Fikes and H.J. Levesque. (October 1983). Krypton: A Functional Approach to Knowledge Representation. *IEEE Computer, Special Issue on Knowledge Representation,* , pp. 67-73.

B.J. Grosz, D.E. Appelt P.A. Martin and F.C.N. Periera. (May 1987). TEAM: An Experiment in the Design of Transportable English Interfaces. *Artificial Intelligence, 32*(2), 173-244.

G. Hobbs. (1985). Ontological Promiscuity. *Proceedings of the 23th Annual Meeting of the Association of Computational Linguistics.* .

Miller, J. R., Hill, W. C., McKendree, J., Masson, M. E. J., Blumenthal, B., Terveen, L., & Zaback, J. (1987). The role of the system image in intelligent user assistance. *Proceedings of INTERACT'87.* Stuttgart.

Rich, E. A., J. Barnett, K. Wittenburg & D. Wroblewski. (July 1987). Ambiguity Procrastination. *Proceedings of AAAI-87.* .

B. Sheils. (1983). Power Tools for Programmers. *Datamation,* , pp. 131-144.