DIAG-NLP: IMPROVING INTELLIGENT TUTORING SYSTEMS WITH NATURAL LANGUAGE GENERATION TECHNOLOGY

BY

DAVIDE FOSSATI
B.S. Equivalent, Politecnico di Milano, Italy, 2001

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2003

Chicago, Illinois

ACKNOWLEDGMENTS

This work was supported by grant N00014-00-1-0640 from the Office of Naval Research. We are grateful to CoGenTex Inc. for making EXEMPLARS and RealPro available to us.

DF

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

ATN	Augmented Transition Network
DBMS	Data Base Management System
CAI	Computer Aided Instruction
CBT	Computer Based Training
ITS	Intelligent Tutoring System
ITSs	Intelligent Tutoring Systems
KB	Knowledge Base
NLG	Natural Language Generation
NLP	Natural Langauge Processing
PDO	Predicate and Direct Object
RU	Replaceable Unit
SP	Subject and Predicate
SQL	Simple Query Language
JDBC	Java Data Base Connectivity
XML	Extensible Markup Language

SUMMARY

The latest generation of Intelligent Tutoring Systems proved to be effective in helping students acquire knowledge. Natural Language Processing technology could be beneficial in improving these highly interactive applications. This thesis describes the latest development of the DIAG-NLP project, a research study conducted to verify the effectiveness of simple Natural Language Generation techniques in Intelligent Tutoring Systems. A rigorous corpus-based methodology has been applied, and multiple prototypes have been implemented and compared. The preliminary results obtained are encouraging and provide some basis for future research.

1. INTRODUCTION

Computers and Information Technology have great potential in supporting instruction and education. Although education is inherently a human activity, computers can help students improve their learning speed and quality, and they can reduce the costs associated with instruction. Natural Language Processing (NLP) techniques can help users that are not technicians interact with machines in an easier and more intuitive way. So, it is possible that NLP technology can have positive effects when applied to computer-supported instruction.

This thesis gives some contribution to this broad and contemporary research field; more specifically, it is about how simple Natural Language Generation (NLG) techniques can be used to improve the output of an existing Intelligent Tutoring System (ITS).

There are two main issues addressed in this thesis. The first one is about how tutoring instructions can be expressed in a fluent and natural way; this issue is partially solved by deducting information from an annotated corpus of human authored texts. The second one concerns the correct implementation of such features in an existing system, and the construction of a framework that could be reused in future work.

The results obtained are consistent with the original expectations and specifications, and they provide some basis for future work on this topic.

Chapter 2 briefly explains some general concepts about Natural Language Generation, Intelligent Tutoring Systems, and NLG applied to ITSs.

Chapter 3 presents the Vivids-DIAG and DIAG-NLP systems, respectively the original ITS subject of our study and the multi-version NLG component designed to improve that ITS.

Chapter 4 presents an information extraction work performed on a target corpus consisting of human tutoring dialogues. The information collected has been used to design the last version of the generation module.

Chapter 5 shows how the information obtained from the corpus analysis has been exploited in the sentence planner of the NLG-enhanced system.

Chapter 6 illustrates the structure of the Language Generation Engine designed for DIAG-NLP3, the last version of DIAG-NLP.

Chapter 7 shows the simple Knowledge Base built to support the language generation process in a domain independent way.

Chapter 8 presents a preliminary system evaluation and draws some conclusive remarks.

Appendix A shows the entire code of DIAG-NLP3 (NLG module only), written in the Java programming language.

Appendix B shows a reference table for the usage of the Language Generation Engine.

2. NATURAL LANGUAGE GENERATION AND INTELLIGENT TUTORING SYSTEMS

This Chapter presents an introduction to Natural Language Generation (NLG), Intelligent Tutoring Systems (ITSs), and NLG technology applied to ITSs.

2.1 Natural Language Generation

Natural Language Generation (NLG) can be defined as the automated generation of natural language text or speech (English, Italian, Chinese, etc.) from formal representations (programming languages, data structures, databases, etc.). This definition is probably too wide to be useful; strictly speaking, almost every computer application contains a NLG portion. For example, one of the easiest computer programs ever written, the "Hello, world!" program, generates a well-formed English sentence. So, it is useful to define at least three classes of Natural Language Generation systems, in order of increasing complexity.

The simplest NLG technique can be classified as *canned text* technique. The natural language text is hard coded in the program and displayed or printed when needed. This is the earliest method used, and it produces excellent results when no or very little flexibility is needed. Up to now, the vast majority of interactive programs use this approach to produce natural language output.

A more complex and powerful technique is called *template filling*. It consists in the composition of multiple sources of text using predefined patterns (*templates*). It is more flexible than canned text and it makes personalization easy. For example, many word processors offer template filling features in order to speed up writing some kind of documents, like business cards or standard-looking web pages. Some template filling systems are so complex that the line between them and full-fledged NLG systems becomes blurred.

Full-fledged NLG systems can synthesize natural language text from domain knowledge represented at a different level of abstraction. Those systems usually process the information to be presented before giving it a textual form. From now on, we will refer to this kind of complex systems just as *NLG systems*, disregarding the simpler ones described before.

2.1.1 **Tasks of a NLG system**

In order to produce appropriate results, a NLG system has several tasks to carry out. These tasks may vary depending on the application, but they can generally be classified in six categories (Reiter and Dale, 1997): *content determination*, *discourse planning*, *sentence aggregation*, *lexicalization*, *referring expression generation*, *linguistic realisation*.

Content determination faces the problem of choosing what information the system has to communicate. Usually the system has a lot of information available, and selecting the right amount of information that the user needs may be a non-trivial task.

Discourse planning is the process where the system organises the order in which to present the information collected. Sometimes a good ordering is crucial to the effectiveness of the communication; for example, a scientific proof may become unreadable or even incorrect if it is presented out of order; an explanation of a process can make the reader spend much less time understanding it if the content is presented in a appropriate order. Besides, sentences can be related in a rhetorical way, in order to improve the overall coherence of the text. A more detailed discussion of such *rhetorical relations* is provided later.

Sentence aggregation refers to the choice of what piece of information can be put together in the same sentence. Producing a distinct sentence for each concept can result in unnatural text; Putting all

together in a single sentence can lead to unreadable text. The right amount of sentence aggregation depends largely on the application domain.

Lexicalization is the process of choosing the right words (*lexemes*) to express the given concept. Sometimes fixed lexemes are associated with concepts, but a more sophisticated lexicalization can be made, for example, if more variety is needed (e.g., to avoid excessive repetition of the same word).

Referring expression generation deals with the choice of right pronouns and other kind of referential expression in a sentence or in a set of multiple sentences. In many natural languages, referring expressions are widely used to improve fluency and avoid repetitions. In order to generate those expressions correctly, contextual factors (like the *discourse history*) must be taken into account.

Linguistic realisation takes care of syntactic, morphological, and orthographic issues. Linguistic realizers must ensure that the sentences generated by the system are correct with respect to the grammar of the target natural language.

The following sections present more details about *discourse planning*, *aggregation*, and *linguistic realization*.

2.1.1.1 **Discourse planning**

According to (Jurafsky and Martin 2000), two mechanisms for building discourse structures are particularly important; they are *text schemata* and *rhetorical relations*.

A *text schema* can be used when all the texts to be generated present a common structure. That structure is represented by a schema. A schema can be modeled with different formalisms, like *flow graphs* (look at Figure 18 in Chapter 5 for an example) or *augmented transition networks*. An augmented transition network (ATN) is basically a state machine with optional transitions and the ability to recursively call itself. For example, consider the ATN depicted in Figure 1. It can be used to model the structure of a text giving a description of a complex object. Using this ATN and a Knowledge Base like the one represented in Figure 2, we could generate a target textual description like the one in Figure 3.

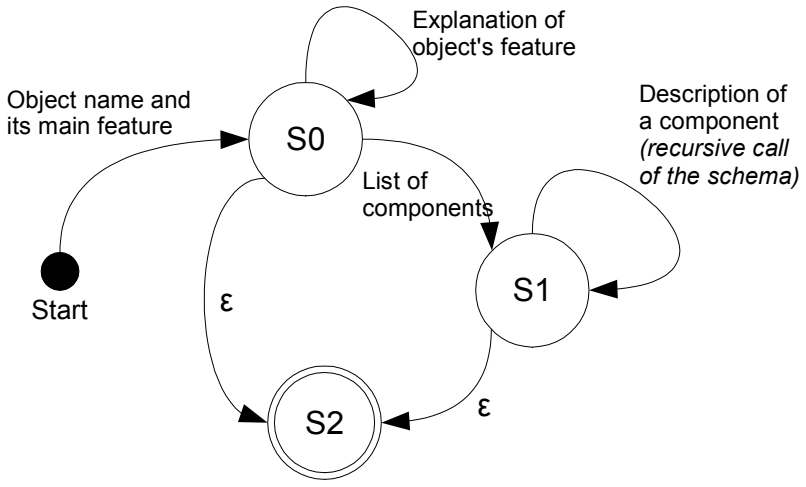

Figure 1. An Augmented Transition Network.

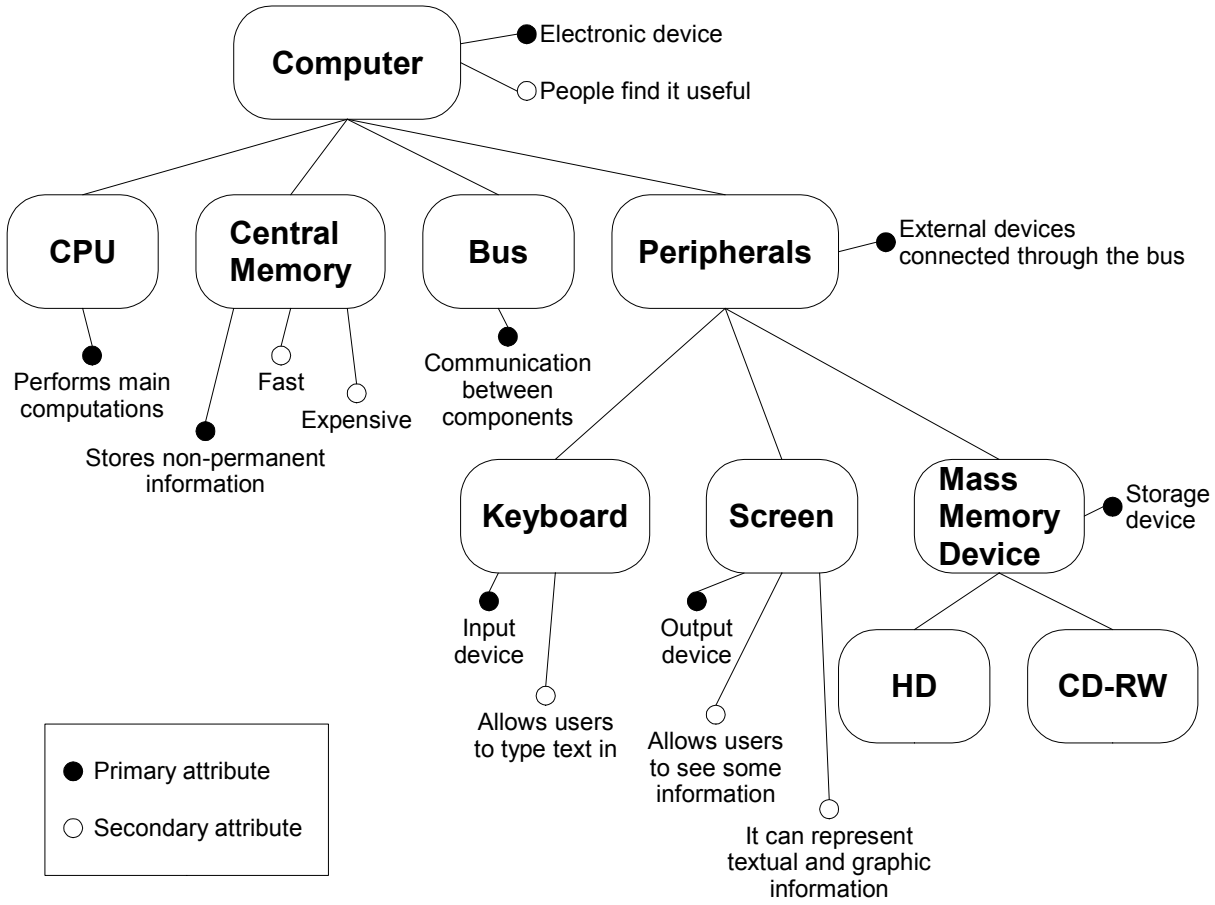

Figure 2. Knowledge Base describing a computer.

A computer is an electronic device. People usually find it very useful. It is composed by a CPU, a central memory, a bus, and some peripherals. The CPU performs the main computations. The central memory stores non-permanent information. It is fast and expensive. The peripherals are external devices connected through the bus. One instance of peripheral is the keyboard, that is an input device.

Figure 3. A sample target text.

Rhetorical relations allow us to represent the relationships that occur between different parts of the same text. A rhetorical relation is usually composed by a nucleus, representing the central segment of text, and one or more satellites, that specify additional properties of the nucleus, according to the relation itself. The following list shows some common rhetorical relations. In the example sentences, nuclei are underlined and satellites are printed in italics.

- *Elaboration*. The satellite tells some additional details about the nucleus. Example: "Mary is a teacher. *She teaches math.*"
- *Contrast*. This is a so called *multi-nuclear* relation, because instead of relating a nucleus and a satellite, it relates sentences that can all be considered nuclei. The *contrast* relation expresses some opposition occurring between two nuclei. Example: "Mary is a math teacher. However, she doesn't know how to add two numbers." In this example the relation is defined by the cue word "however"; without this word, the entire sentence would appear incoherent ("Mary is a math teacher. She doesn't know how to add two numbers.")
- *Condition*. The satellite presents a prerequisite for the nucleus to occur. Example: "Mary can take a high grade in her exam *only if she studies a lot.*"
- *Purpose*. The satellite expresses the goal for which the nucleus is performed. Example: "Mary studies a lot *in order to get a high grade in her exam.*"
- *Sequence*. In this multi-nuclear relation, the set of nuclei are realized in succession. Example: "First, you buy a ticket; then, you get on the train."
- *Result*. The situation expressed in the nucleus results from what is expressed in the satellite. Notice that this is different from the *condition* relation. Example: "I take my umbrella *because it's raining.*"

Rhetorical relations can be composed in complex structures. Figure 4 shows an example of text with multiple rhetorical relations, represented in a graphical notation.

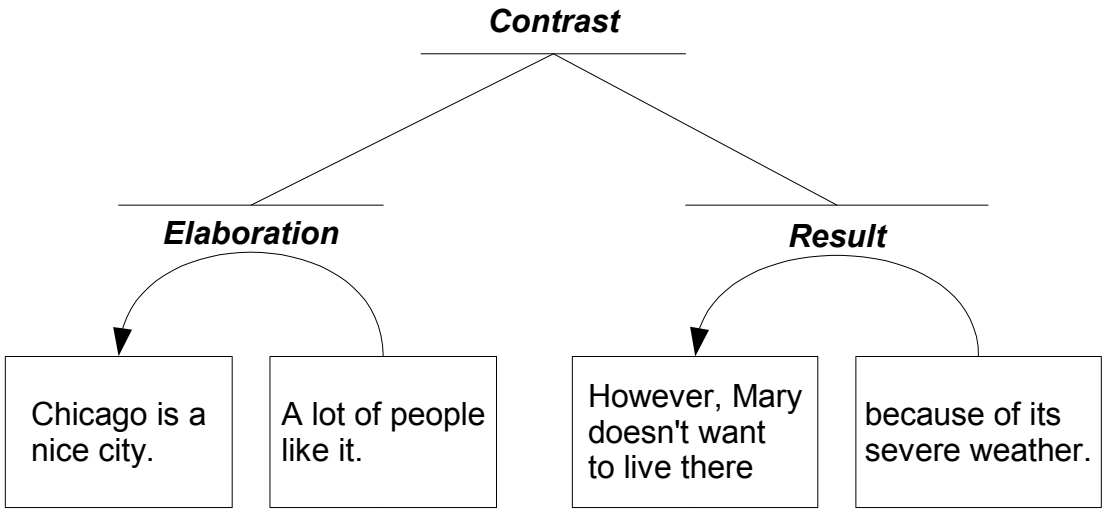

Figure 4. Rhetorical relations.

2.1.1.2 **Aggregation**

According to (Reape and Mellish 1998), we can define aggregation as "the combination of two or more linguistic structures into a single linguistic structure which contributes to sentence structuring and construction." Generally speaking, the goal of aggregation is to reduce some linguistic redundancy in order to improve text fluency. According to (Dalianis 1996), we can classify four main types of aggregation: *syntactic aggregation*, *elision*, *lexical aggregation*, and *referential aggregation*.

- *Syntactic aggregation* removes some redundant information applying rules concerning only the syntactic level of the text. Examples of those rules are the *Predicate and Direct Object (PDO)* aggregation and the *Subject and Predicate (SP)* aggregation. Syntactic aggregation rules preserve enough elements of the original text to carry the original information explicitly.

Example of PDO aggregation:

John is a professor. Mary is a professor.

↓

John and Mary are professors.

Example of SP aggregation:

Mary likes cakes. Mary likes ice-creams.

↓

Mary likes cakes and ice-creams.

- *Elision* removes some explicit information from the text, but the information removed can be inferred by the reader, so it is left there implicitly.

  Example:

  - Where did you buy those shoes?

  - I bought them in an outlet store.

  ↓

  - Where did you buy those shoes?

  - In an outlet store.

- *Lexical aggregation* (or *functional aggregation*) replaces a set of elements with a new one representing some shared properties of the original items. We distinguish between two types of lexical aggregation: *bounded lexical aggregation*, where the replacing element completely represents the entire set, and *unbounded lexical aggregation*, where the aggregated elements are not completely represented by the aggregator. Unbounded lexical aggregation causes some loss of information.

Example of bounded lexical aggregation:

John works on Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday.

↓

John works the entire week.

Example of unbounded lexical aggregation:

Mary bought meat, fish, and vegetables at the grocery store.

↓

Mary bought some food at the grocery store.

- Referential aggregation replaces some redundant explicit information with a pointer to it, such as a pronoun or a referential expression.

  Example:

  John, Peter, and Paul are going to a party tonight.

  John, Peter, and Paul are going to have fun.

  ↓

  John, Peter, and Paul are going to a party tonight.

  Those guys are going to have fun.

2.1.1.3 **Linguistic realization**

There are many approaches used to implement linguistic realization (otherwise called *surface realization*); three of them make use of *systemic grammars*, *functional unification grammars*, and *meaning-text grammars*.

Systemic grammars are built on the basis of a linguistic theory called *systemic functional linguistics* (Halliday 1985). According to this theory, sentences in natural languages can be represented with a set of *functions*; systemic grammars describe how these functions are mapped to surface forms. A

realization process driven by a systemic grammar can be viewed as a series of choices; the grammar is traversed progressively in order to collect a set of realization statements that are coherent with the input data and the knowledge stored in a knowledge base. For a more extensive explanation, see (Jurafsky and Martin 2000). An example of implemented system using this approach is the linguistic realizer KPML (Bateman 1996).

Functional unification grammars (Kay 1979) are represented as big *feature structures*; storing the input to be realized in a feature structure consistent with the grammar, it is possible to use the *unification algorithm* to carry out almost the entire realization process. For a detailed description of feature structures, the unification algorithm, and its usage in the surface realization task, see (Jurafsky and Martin 2000). An important example of system using unification is the FUF/SURGE package (Elhadad 1993; Elhadad and Robins 1996).

Meaning-text grammars are based on *Meaning-Text Theory* (Mel'cuk 1988). According to this theory, languages can be represented in a model with seven layers, spacing from *meaning* to *phonology* (Figure 5). Each layer has its own structural representation; In fact, a meaning-text grammar is a set of grammars designed to convert the linguistic content from a layer to the next one. An example of a realizer using this approach is RealPro (Lavoie and Rambow 1997). RealPro starts the realization task from the *Deep Syntactic level* of the meaning-text model, down to a textual representation in natural language.

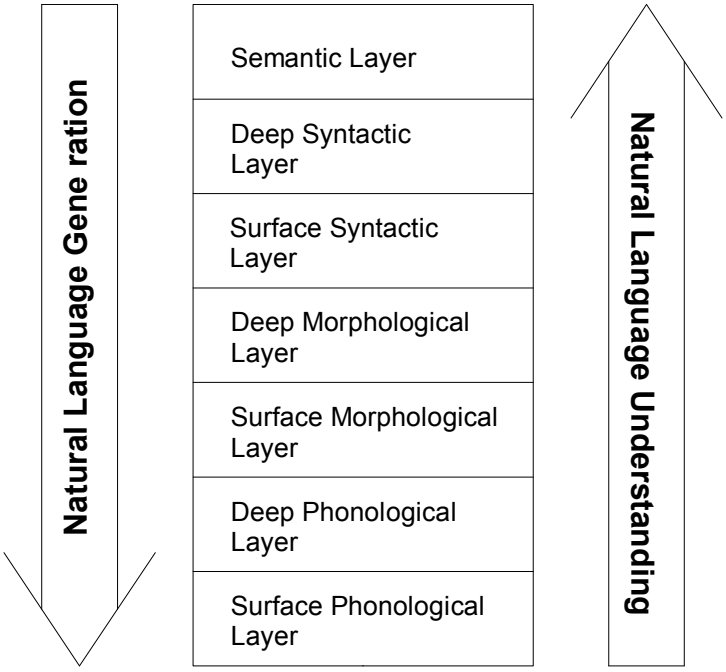

Figure 5. Layering in Meaning-Text Theory

2.1.2 **Architecture of a NLG system**

Of course, NLG systems can have a wide variety of architectures; a good trade-off between complexity and design quality is a three-tier architecture like the one shown in Figure 6. Such a system is composed by three modules: a *text planner*, a *sentence planner*, and a *surface realizer*. These modules are connected with a pipeline.

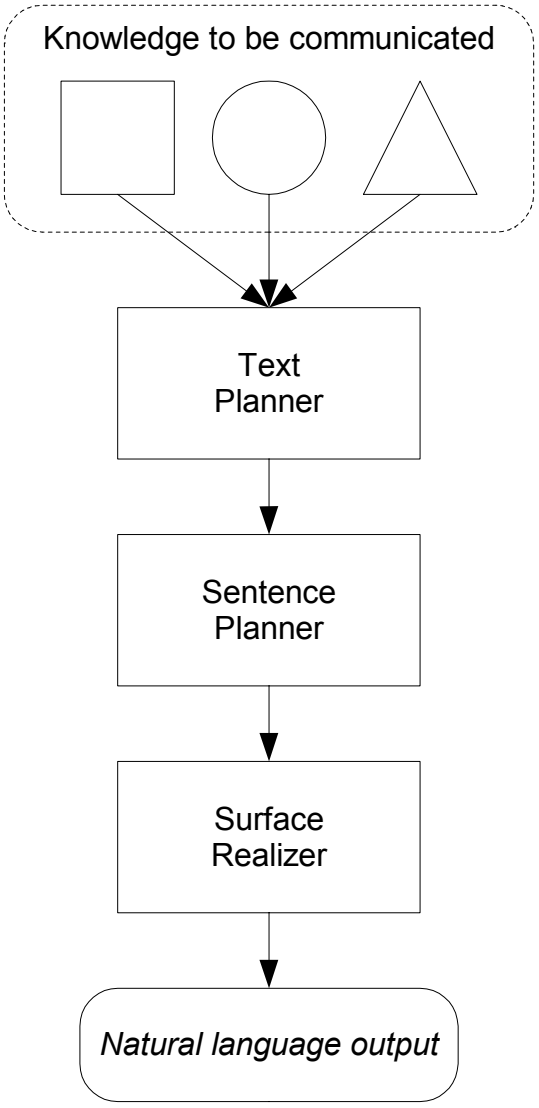

Figure 6. Architecture of a NLG system.

The *text planner* carries out the tasks of content determination and discourse planning. This module works at a very high level of abstraction. The information can be processed here in terms of *concepts*, *entities*, *relations*, and *messages*. The system may need access to sophisticated Knowledge Base Systems to perform deep reasoning and inference on this knowledge. Usually this module is the most dependent on the application domain, so it is difficult to find commonly accepted standards (such as languages and formalisms) at this level.

The *sentence planner* performs the tasks of sentence aggregation, referring expression generation, and lexicalization (sometimes lexicalization is moved to the next module). So, the sentence planner works on the overall structure of the sentences, refining more details left undecided by the text planner. While the text planner makes decisions mainly about the *content* of the communication, the sentence planner makes high impact choices on the communication *style*. With this approach, it is possible to build a system that produces texts with different styles (e.g., formal and informal) by applying different sentence planning routines to the output of the text planner.

The *surface realizer* deals with the low-level grammar of the target natural language. It is mainly a translator between a formal structure (e.g., a tree-like structure) representing the sentences and a plain text (or formatted text) document. Notice that, at this level, almost all the communicative choices have already been made by the previous modules. Moving the lexicalization task down to this module can make it easier to produce output texts in multiple languages.

2.2 **Intelligent Tutoring Systems**

Intelligent Tutoring Systems (ITSs) are systems that provide personalized tutoring or instruction. Their main feature is the ability to adapt their teaching to the particular needs of different learners. ITSs can be thought as an evolution of *Computer Based Training (CBT)* and *Computer Aided Instruction (CAI)* systems, in which the instruction was presented as a series of predefined steps. (Beck et al. 1996; Broberg 2000; Graesser et al. 2001).

2.2.1 **ITS knowledge**

To achieve the desired goals, an ITS needs to process three kind of information:

1) knowledge about the domain to be taught;
2) knowledge about the learner;
3) knowledge about teaching strategies.

The *knowledge about the domain* represents the source of the topics to be taught to the learner. Building a good knowledge base makes the designer face the classic issues of Knowledge Engineering, such as interaction with human experts, knowledge representation models, and scalability. All of these issues are still subject of research.

The *knowledge about the learner* is the key for providing individualized instruction. The system needs to build a *learner model* and update it while the tutoring process goes on. The learner model represents the beliefs of the system about what the student knows or does not know about the domain. It can be very difficult for the system to acquire such knowledge accurately; usually it is derived from the evaluation of the student's performance in solving problems or answering questions. The two main methods to represent the learner model are based on *overlay models* and *Bayesan networks*. In *overlay models*, the student's knowledge is supposed to be a subset of the system's knowledge, possibly augmented with some “buggy” knowledge, consisting of the student's misconceptions about the domain

(Figure 7). Using this approach, the ITS will present the topics in such a way that the learner's knowledge will eventually match the system's knowledge. In *Bayesian networks*, the student's knowledge is represented probabilistically; each node in the network has a value representing the probability that the learner knows some piece of knowledge; these values are then updated accordingly to the interactions with the tutor.

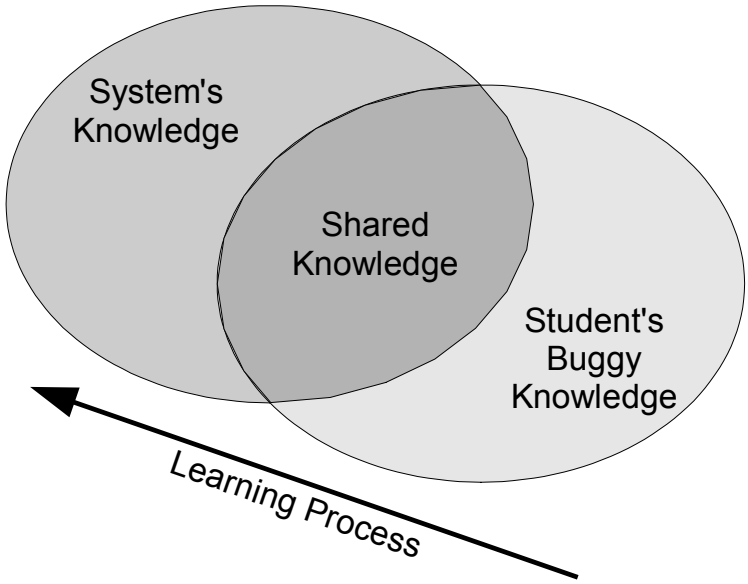

Figure 7. Overlay pedagogical model.

The *knowledge about teaching strategies* is used to decide how to present the material to the students. These decisions are mainly about the *topics* to be explained, the *problems* to be assigned, and the amount of *feedback* to be given. The *topic selector* affects, for example, the choice of presenting new information or reviewing something the student already saw. The *problem generator* must produce exercises adequate to the actual skills of the student. The *feedback generator* provides hints to the students that have trouble with the topics or problems proposed; a good ITS can provide feedback tailored to the ability of the learners, for example providing subtle advice if the student is skilled or direct hints if the student is in serious trouble. All of these choices, of course, must be made accordingly to the student model too. Teaching strategies could be built on the basis of research and studies in the field of psychology and cognitive sciences. Tutors built following these guidelines are often called *cognitive tutors*.

2.2.2 **Architecture of an ITS**

Up to now, there are no widespread standards in the architectural design of an ITS; many of them are monolithic programs that incorporate all the functions needed by the application. However, it is useful to think of an ITS as composed by five separate modules: *pedagogical module*, *student model module*, *domain knowledge module*, *expert model module*, and *communication module* (Beck et al. 1996). This architecture is represented in Figure 8. The pedagogical module encapsulates the knowledge of teaching strategies and performs the choices about what to present; it takes as input the student model stored and maintained in the *student model module*, and the domain information stored in the *domain knowledge module* and in the *expert model module*. The *expert model module* contains how the domain knowledge is represented by someone expert in the application domain; so, it is related to domain knowledge but it does not contain the domain knowledge data. The *communication module* has to provide the interaction with the student, using graphical or textual interfaces, dialogues, and so on.

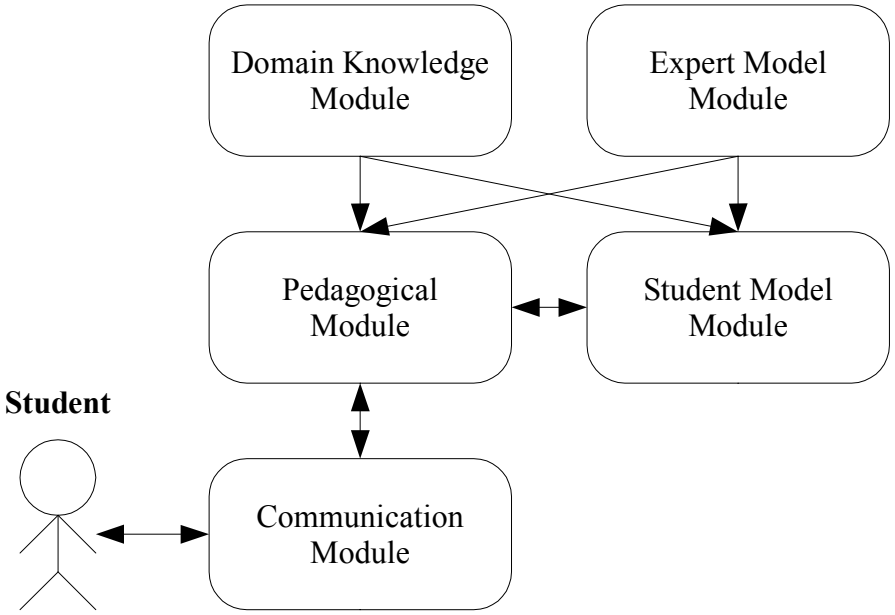

Figure 8. Architecture of an Intelligent Tutoring System.

2.2.3 **Advantages of ITSs**

There are three major issues that make ITSs interesting in "real world" educational contexts: *cost efficiency*, *learning efficiency*, and ITSs potential for *bridging distances* in educational environments (Broberg 2000).

Cost efficiency (in a learning context) is the ratio of knowledge acquired over the total amount of money invested. ITSs can significantly improve this ratio for several reasons. There are multiple sources of cost for a company in order to provide instruction for its employees. A company usually removes learners from production during education, and unproductive people represent a cost for a company. People usually need to be sent away for education, involving costs for travel and living expenses. Finally, teachers and equipment represent a considerable cost. Building an ITS is not cheap: it has been estimated that 100 hours of system development result in one hour of instruction (Murray and Woolf 1992). Nevertheless, the adoption of an ITS can help limit the sources of cost described above in many situations.

Learning efficiency is the ratio of knowledge acquired over the total time spent. Several researchers have shown that ITSs can improve learning efficiency significantly with respect to a standard classroom environment (Shute and Psotka 1994). Since learning time is often a source of cost, improving learning efficiency has positive effects on cost efficiency too.

The potential for *bridging the distances* in education is a promising feature of ITSs. There are four main distances in a learning context: *distance in space*, *distance in time*, distance *in the way to study*, and *distance to relevant material*.

- *Distance in space* is one of the classical definitions of distance; it refers to the fact that students may be located in different areas.
- *Distance in time* means that learners can study in different time slots.

- *Distance in the way to study* means that students may have different preferences about how to perform the learning process; for example, some students prefer to review a large quantity of theoretical material before applying it, while others prefer to learn theory from examples and exercises.
- *Distance to relevant material* refers to how difficult it is for a student to have access to useful material to study; such difficulties can arise for several reasons, like the availability of textbooks and papers, the languages in which they are written, and so on.

Traditional classroom environments tend to eliminate all of these distances; students are located in the same room at the same time, they all study the curriculum built by the teacher, and teachers usually choose textbooks and other material. However, there are advantages in maintaining some distances, so that student can exploit their learning potential without being bound by tight constraints; ITSs can help manage such distances without having to eliminate them like in traditional classrooms.

ITSs have proven to perform very well in many situations. A possible measure of the quality of education is the *learning gain* of students. The *learning gain* is the difference between a student's performance in a test before and after receiving instruction. Some researchers show that the learning gain of students instructed with ITSs is usually located halfway between that of students educated in a traditional classroom setting and that of those that received personalized human tutoring (Anderson et al. 1995). For all these reasons, ITSs have the high potential to become widely adopted in many learning situations.

2.3 Natural Language Processing for ITSs

2.3.1 Advantages of NLP for ITSs

It has been said that current ITSs have a performance, in terms of student's learning gain, that scores between traditional classroom instruction and personalized human tutoring. So, active research is being conducted in order to bridge this gap. Among the possible improvements under investigation, the usage of Natural Language Processing (NLP) techniques looks promising. In fact, there are several reasons that make us think that improving natural language interfaces for ITSs could be beneficial.

1) Human tutors often exploit pedagogical techniques to help students discover knowledge by themselves, instead of just presenting the topics for students to learn. One of these techniques is based on natural language, specifically on *conversational dialogues*. In a conversational dialogue, a tutor interacts with the student asking him/her questions and providing some feedback based on his/her answers. Through this process, the student can gradually improve his/her knowledge, adding new pieces of information and fixing misconceptions.

2) Using natural language appropriately, human tutors do not teach only domain concepts, but they also teach students *how to talk about the topics they learn*. This skill is very important, for example in team working; its development requires a knowledge deeper than the mere acquisition of notions.

3) Learning new formalisms, even graphic ones, can cause some overhead for students. Natural language has the advantage of being easy and immediate for learners to understand and use.

Examples of successful ITSs using natural language and conversational dialogue are CIRCSIM-Tutor (Evens et al. 1993; Glass 1999; Evens et al. 2001) and AutoTutor (Graesser et al. 2001).

2.3.3 **Methodology**

The main problem in designing a Natural Language Interface for an ITS is that we do not know exactly what features of natural language have a positive impact on teaching. To to overcome this obstacle, an accurate methodology should be used (Di Eugenio 2001; Di Eugenio et al. 2003). Such a methodology can be divided into five steps: *data collection*, *data annotation*, *annotated data analysis*, *system implementation*, and *system evaluation*.

The *data collection* step consists of gathering a corpus of human authored documents that present features that may be relevant with respect to the system being built. An example of such a corpus is a collection of student-teacher dialogues. The corpus must be big enough to contain some variety of linguistic examples, and it must be constrained enough in order to guarantee some homogeneity with respect to relevant domain characteristics.

The *data annotation* phase consists of the definition of a relevant set of features (coding scheme), followed by the formal annotation of the corpus text with this additional information. This operation is necessary because natural language text is complex and highly unstructured. Annotation adds structured content to the corpus, making it possible to perform computations on it more easily.

The *annotated data analysis* consists in the extraction of linguistic models from the annotated corpus. A set of hypotheses, formulated during the coding scheme definition, can be verified here; for example, such analysis can reveal that some linguistic patterns are preferred over others in tutors' answers.

The *system implementation* phase consists in the realization of the features discovered before in the form of algorithms and executable systems. For research purposes, such systems can be built in multiple

versions, each one implementing different features. This makes it possible to evaluate the impact of each feature separately and compare them.

The *system evaluation* step tries to check whether the implemented features have a positive effect on the system's ultimate goals or not. An effective system evaluation is very difficult to run, because there are many different factors that influence the system's final performance, and it is not easy to identify original causes from joint effects.

The methodology described above can be very time consuming, especially for small groups of researchers. However, it is likely that a system built following these guidelines can be useful in providing some directions to future ITS research and development.

3. VIVIDS-DIAG AND DIAG-NLP SYSTEMS

After the literature review presented in Chapter 2, let us talk now about the system that is the main subject of this study. At the present time, there are four versions of the system: Vivids-DIAG (the original one), DIAG-NLP1, DIAG-NLP2, and DIAG-NLP3 (the NLP-enhanced versions). The following sections will present them.

3.1 **Vivids-DIAG**

Vivids-DIAG, a combination of the Vivids authoring environment (Munro, 1994) and the DIAG framework (Towne 1997a; 1997b), is a tool designed to build Intelligent Tutoring Systems (ITSs). A typical ITS built with Vivids-DIAG provides a student with a visual representation of a relatively complex system. This system is modeled with a set of components called *replaceable units* (RU) and a set of *indicators*. The ITS simulates the failure of one or more replaceable units. The student has to troubleshoot the system, replacing all the broken replaceable units. To find out which replaceable units are faulty, the student can read the values stated by some indicators. A skilled student will successfully correlate the symptoms stated by those indicators with the malfunctioning components of the system. The subject of our study is a particular instance of DIAG-based ITS, the *Home Heating System*; it is a demonstrative ITS found in the DIAG package. Figure 9 and Figure 10 are two screenshots of the Home Heating System.

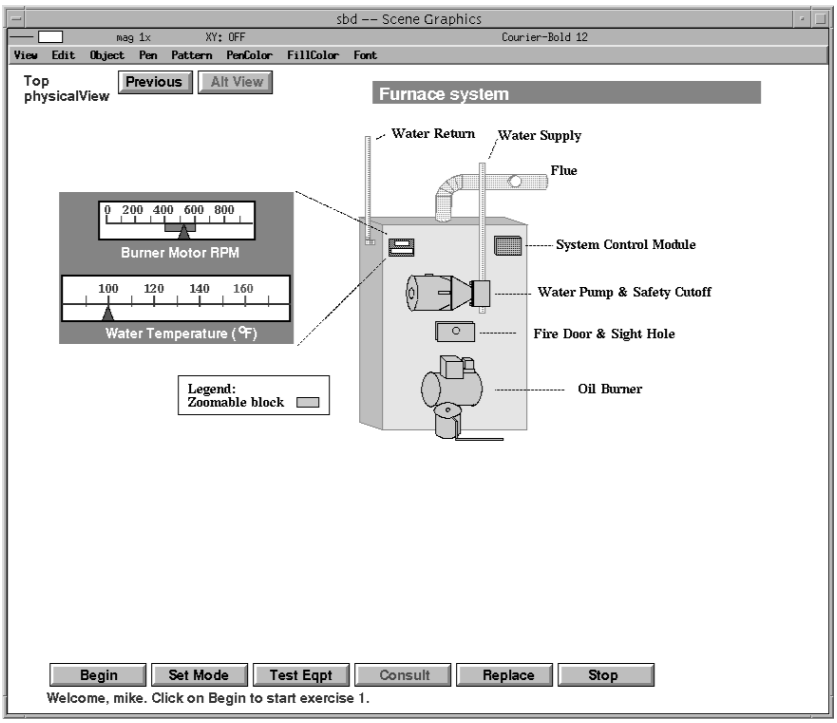

Figure 9. The furnace screen in the Home Heating System.

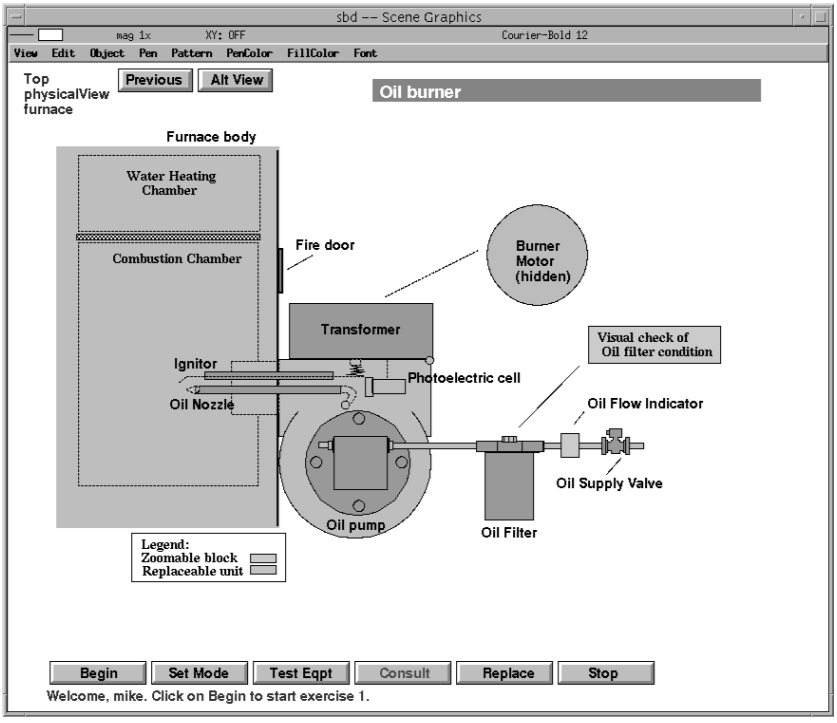

Figure 10. The oil burner screen in the Home Heating System.

To help the student acquire the skills required to successfully debug the faulty system, DIAG can provide suggestions based on the investigation path of the student, i.e., the parts of the system he/she saw and the indicators he/she read during the current exercise. This advice, given in natural language, answers four types of queries: *Consult RU*, *Consult Indicator*, *Consult Debrief*, and *Show Suspicious RUs*. To answer a *Consult RU* query, DIAG provides an indication of the likelihood that a given RU is the cause of the fault. A *Consult Indicator* query asks for the correlation of a given indicator reading with a set of replaceable units. The *Consult Debrief* and *Show Suspicious RUs* queries provide summary details about the current exercise; the last two kinds of queries are mostly unused in our experiments. Student interact with the system graphically; all of the queries are given by clicking on the appropriate query button.

A sample result of a Consult Indicator query is shown in Figure 11. From a linguistic point of view, the output of the ITS appears poor and monotonous. There is a lot of repetition that makes the text difficult to read.

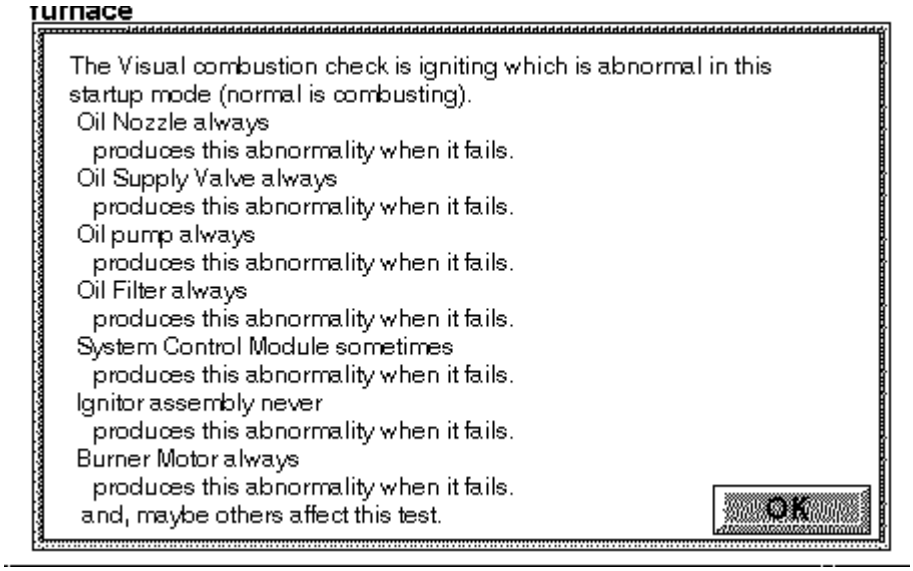

Figure 11. Consult Indicator answer of Vivids-DIAG.

From an architectural point of view, DIAG does not map exactly to the architecture presented in the previous chapter (Figure 8). For example, there is no distinct pedagogical module.

3.2 **DIAG-NLP1**

The first version of the DIAG-NLP project, carried out in the NLP Lab at the University of Illinois at Chicago, aimed to improve the raw output of DIAG to make it more fluent. To do that, the researchers decided to perform simple clause aggregations on the original output. Clause aggregation is a technique used to combine different elements from multiple sentences to a single sentence, taking into account only the syntactic roles of the elements (See *syntactic aggregation* in Section 2.1.1.2). For example, the sentences "Oil pump always produces this abnormality when it fails" and "Oil filter always produces this abnormality when it fails" could be aggregated in a sentence like "Oil pump and oil filter always produce this abnormality when they fail." This is an example of Predicate and Direct Object aggregation (Section 2.1.1.2).

The implementation of this system faced several technical difficulties. The main issue was how to interface a new language processing module to the Vivids-DIAG system. Vivids-DIAG was a stand-alone tool, without the necessary interface for add-on modules. That problem was solved using the Vivids-DIAG internal script capabilities. Some code was written to make DIAG able to dump log files and establish socket connections, in order to send and receive commands from an external program. This external program traps DIAG consult requests, reads the information DIAG would have provided to the user, processes it, and sends it back to DIAG to display. A model of this system is depicted in Figure 12.

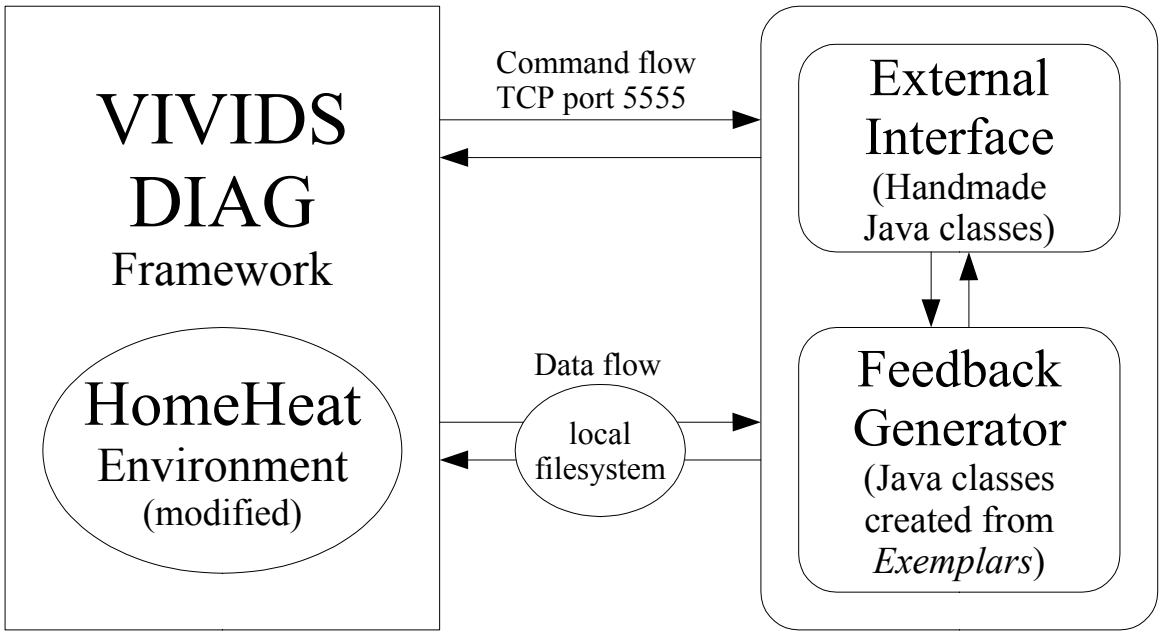

Figure 12. Architecture of DIAG-NLP1.

A sample output from DIAG-NLP1 is shown in Figure 13. As we can see, this text is much more fluent than the original one. The main technique used here is *functional aggregation*; the replaceable units mentioned are grouped by the system modules that contains them (in the example shown below, "Oil Burner" and "Furnace System"), and they are grouped by the relationships of the replaceable units with the observed malfunction ("always produces," "never produces," etc.). Another technique used is *text formatting*, obtained by adding blank spaces, vertical lists, and punctuation to make the text appear more clean and readable. An experimental evaluation of this system has been carried out, and the results showed that the modified system performs better than the original one on the whole (Di Eugenio et al., 2002).

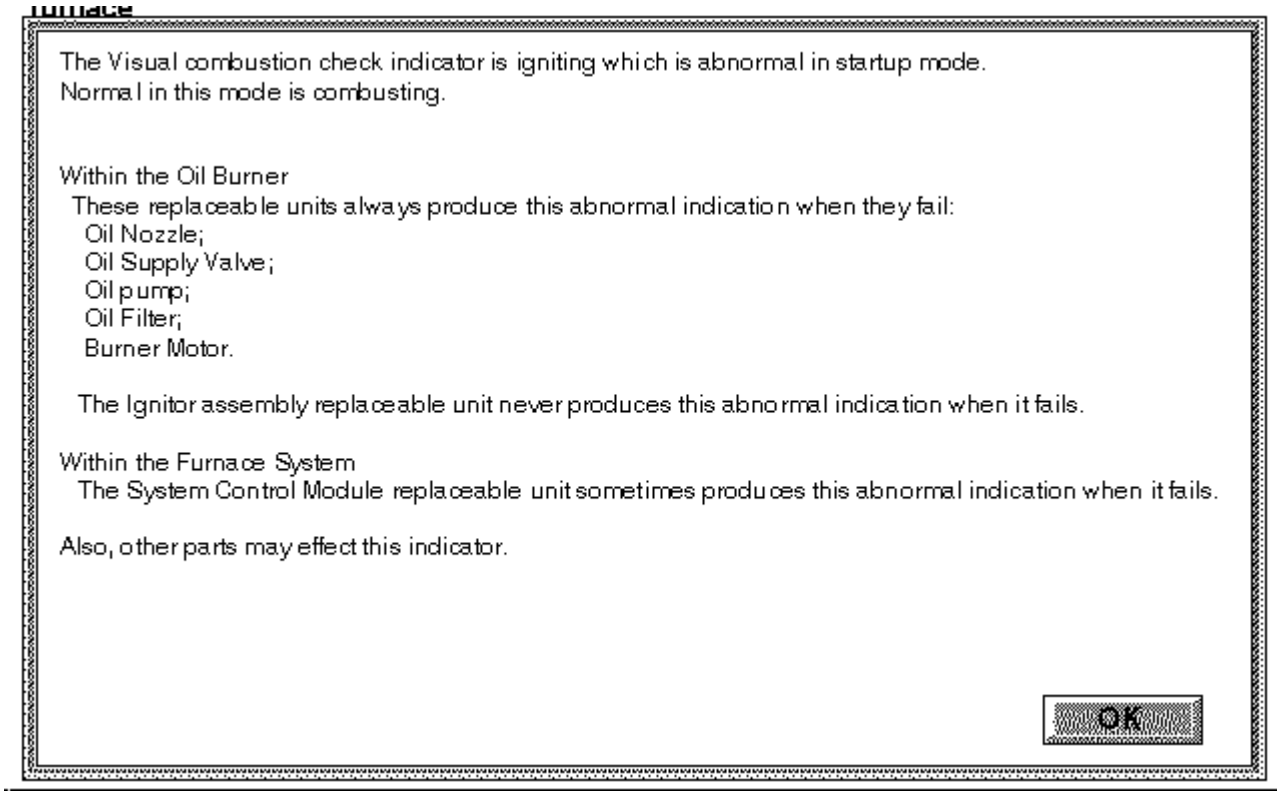

Figure 13. Output of DIAG-NLP1 answering a Consult Indicator query.

3.3 **DIAG-NLP2**

DIAG-NLP2 is a modified version of DIAG-NLP. It has been developed at the Computer Science Department at the University of Wisconsin-Parkside. It adds two main features to DIAG-NLP1, namely some *rhetorical relations* and *referential expressions*. Rhetorical relations aim to make the logical coherence of sentences more explicit. For example, the system can emphasize the contrast between a sentence and the previous one, using the cue phrase *in contrast*. Referential expressions are used to improve conciseness. Pronouns are used to avoid the repetition of nouns or sentences. To implement these features, an external knowledge base was built using the SnePS representation system (Shapiro and Rapaport, 1992). This knowledge base allows the system to reason about entire propositions, making it easy to generate referential expressions and some rhetorical relations.

A sample output of the DIAG-NLP2 system is shown in Figure 14. Notice that the aggregation is performed as in DIAG-NLP1; the pronoun "this" is used as referential expression; the *contrast* rhetorical relation is used in the last paragraph. Comparing this figure to the previous one (the DIAG-NLP1 sample output) , we can see that the information expressed is the same, but the groups of units are presented in a different order. The conciseness is improved by the fact that the pronoun "this" avoids the repetiton of the expression "abnormal indication"; the list of units in a certain group are put in a single sentence (e.g., "this is always caused if *the burner motor, oil filter, oil pump, oil supply valve or oil nozzle* has failed"). The text formatting is also different.

The Oil Flow indicator is not flowing in startup mode.
This is abnormal.
Normal in this mode is flowing.

Within the Furnace System,
this is sometimes caused if
the system control module has failed.

Within the Oil Burner,
this is never caused if
the ignitor assembly has failed.

In contrast, this is always caused if
the burner motor, oil filter, oil pump, oil supply valve or oil nozzle has failed.

Figure 14. Output of DIAG-NLP2 answering a Consult Indicator query.

3.4 **DIAG-NLP3**

DIAG-NLP3 system represents my personal contribution to the DIAG-NLP project. Referring to the general methodology described in Chapter 2, I worked on the third and fourth steps, namely the *annotated data analysis* and the *system implementation*. DIAG-NLP3 does not build on DIAG-NLP2; its design and implementation start directly from DIAG-NLP1.

There were two main goals to achieve in this new release of the system.

1) Neither DIAG-NLP1 nor DIAG-NLP2 had a reference corpus from which to extract information. After the release of DIAG-NLP2, such a corpus had been collected and annotated, allowing a new version of the system to be built based on it.
2) It was necessary to improve the overall design of the system, in terms of software engineering issues.

Given these goals, the development of DIAG-NLP3 focused on three main areas of research.

1) Mining information from *the DIAG corpus* (Di Eugenio et al., 2002) and exploiting it to improve system's feedback.
2) Building a new language generation engine.
3) Building a knowledge representation system that makes it easy to handle aggregation and other kinds of semantic processing.

All of these points will be discussed in detail in the next chapters.

4. MINING THE DIAG CORPUS

The DIAG-NLP project was conceived to improve the natural language output of the DIAG system. However, this goal is very difficult to state with precise and unambiguous terms. Who can decide whether a text is better than another one, and how? Even before that, what does it mean that a text is "better" than another one?

Of course there are not unique satisfactory answers to those questions. A text considered good in a given context could prove inadequate in another context. To find out what could be appropriate in our context, a series of experiments have been carried out, in order to collect useful data to analyze (Di Eugenio et al., 2002).

4.1 **Description of the experiments**

Twenty-three special DIAG sessions were completed by some students. The system was modified such that a human tutor replaced the machine generated answers with his own answers. Students and tutors could not see nor talk to each other, all their interactions passed through the DIAG interface. The students used DIAG exactly as usual. They knew that a human tutor was there, but from a behavioral point of view the only difference was that the answers they got were generated by a human instead of a computer. The human tutors were monitoring all the student's interactions with the system. When a consult request was performed, the tutor (but not the student) saw the answer DIAG would have provided, and replaced it with his/her own response. In this way, the tutor was not enforced to use the information DIAG would have given, but he/she could use it if he/she wanted.

Complete log files were recorded. These log files contain all the requests, the information DIAG would have provided, and the human answers. Figure 15 shows a sample turn taken from a log file.

```
<consult type="ConsultIndicator" prob="1">
<diag-data>
<line id="t10_r1">***INDICATORS***</line>
<line id="t10_r2">Water Temperature Gauge  State=100  Normal=100
Mode=startup</line>
<line id="t10_r3">***REPL UNITS***</line>
<line id="t10_r4">Oil Nozzle.............  no effect</line>
<line id="t10_r5">Oil Supply Valve.......  no effect</line>
<line id="t10_r6">Oil pump..............  no effect</line>
<line id="t10_r7">Oil Filter.............  no effect</line>
<line id="t10_r8">System Control Module..  no effect</line>
<line id="t10_r9">Ignitor assembly.......  no effect</line>
<line id="t10_r10">Burner Motor...........  no effect</line>
<line id="t10_r11">others.................</line>
</diag-data>

<tutor-resp id="t10_turn_1">
Since the oil is not flowing, you know that there is something wrong with the oil.
Since the valve is fine, check where the oil "comes out" from.
</tutor-resp></consult>
```

Figure 15. Example of turn in the collected log files.

4.2 **Log files annotation**

The log files mentioned above contain a lot of unstructured information. In order to effectively exploit this information, a large work of *annotation* was required. The annotators chose to use XML (eXtensible Markup Language) to add structured information to unstructured text. The choice of what to annotate was a very delicate point, because an unwise annotation would lead to poor results in any subsequent information extraction.

Three main areas of information have been captured by the annotation; these are the *home heating domain*, the *diagnosis process*, and the *tutoring style*. TABLE I shows a list of the types of annotations performed, cited with the XML tags used to perform them. All tags have multiple attributes to better specify several annotation properties; they are omitted in this table. Those attributes provide vital information about the communication properties of the dialogues. For a complete reference, see (Glass et al., 2002).

TABLE I. TAGS USED FOR THE ANNOTATION.

Tag name	***Description***	***Examples***
<indicator>	It marks the presence of an indicator in the text.	Click on *the oil flow indicator.*
<indication>	It marks the presence of an indicator together with some state information.	Note *the low water temperature* in the previous screen.
<ru>	It marks a replaceable unit in the text.	*The oil pump* is likely pumping oil.
<operationality>	It marks a replaceable unit together with some state information.	*The oil pump is likely pumping oil*.
<linguistic_aggregation>	It indicates the syntactic combination of several items.	Some indications you have seen that the burner motor is working: *the burner motor rpm gauge is normal, the oil is flowing, it is combusting*.
<aggregate_object>	It indicates an aggregate object, i.e. an object composed by several parts.	There are no problems in *the oil burner*.
<probsolving>	It marks a direct hint to the student from the tutor.	*Check to make sure the water temperature is correct*.
<domain_knowledge>	It indicates whether the tutor tells the student some knowledge about the domain.	*The water temperature safety cutoff valve sets a limit on the temperature of the water*.
<judgement>	It marks a tutor's judgement about the student's actions.	The nozzle is a *very good guess*.
<continuity>	It indicates whether a statement logically depends on a previous one, given in a previous turn.	[Turn 1] The room water supply valve is a very strong suspect. [Turn 2] The water temperature safety cutoff valve is *another suspect*.
<included>	It indicates that some information DIAG would have provided to the student is present in tutor's answer.	[DIAG]: ***INDICATORS*** Burner motor RPM gauge State=525 Normal=525 [Tutor]: Since the burner motor RPM gauge appears to be normal, the burner motor is working properly.

Tag name	***Description***	***Examples***
<excluded>	It indicates that some information DIAG would have provided to the student is not present in tutor's answer.	[DIAG]: ***INDICATORS*** Water Temperature Gauge State=100 Normal=100 Mode=startup [Tutor]: Since the oil is not flowing, you know that there is something wrong with the oil.
<contradicted>	It indicates that some information DIAG would have provided to the student is contradicted by tutor's answer.	[DIAG]: *The Water temperature safety cutoff valve is a very poor suspect*. [Tutor]: Try changing the cutoff valve, if not the photoelectric cell or the transformer.

4.3 **Using the annotated log files to extract information**

Thanks to the annotation work done on the log files, it is possible to carry out some data analysis more systematically and accurately than simply reading the original texts. Generally speaking, there are two main ways to perform this task, *human driven statistical techniques* (i.e., finding out some patterns in the corpus and trying to figure out if they have statistical support) and *machine learning*. At this early stage of our research, machine learning is very difficult to use, so a human driven interpretation is required.

Unfortunately, there were some major issues that made this data analysis extremely difficult and not completely enlightening.

1) Our log files corpus is quite small. Even though the number of dialogues collected is not negligible, the number of different linguistic phenomena occurring in those log files is so high that the number of occurrences of each phenomenon is small.

2) There are a lot of "non meaningful" turns in the collected log files, like empty, test, interrupted, or garbled turns. Although the total number of student requests collected is 423, the number of meaningful tutor's answers is 274. Sometimes, there is also evidence of biased student's or tutor's behavior. For example, some students used to click randomly on the request screen, making requests that are not logically related one with the other. Sometimes tutors "got tired" of the session going on and gave the students too explicit advice, in order to finish quickly. This leads to the problem whether those log files represent a realistic approximation of student's interactions with an Intelligent Tutoring System or not. This is a common issue in data collection experiments.

3) For technical convenience, the log files were annotated using *standoff markup* instead of standard markup. In standoff markup, the data to be annotated and the annotation tags are stored in

different files. This was a very elegant choice, because multiple annotations can be done on a single log file without the need to replicate the log file itself for every different annotation. However, standoff markup XML files are very difficult to query. The reason is that XML standoff markup is not yet a standard, so there is a lack of tools available to manipulate them. In fact, the annotation job was performed with a dedicated tool, called *MUP* (Glass and Di Eugenio, 2002).

To query the annotations, I had to do some "craftsman" work, like splitting the files, joining, interleaving, and filtering them. The most useful tool I have found is *sggrep*, a version of the popular *grep* utility able to handle XML files. This tool can be found as part of the LTXML library, that can be found on the Internet.

Because of these reasons, full statistical methods could not be used on the annotated log files corpus. Several statistics were collected and used as guidelines in the design of the new sentence planner of the system.

Let us examine now some of the statistics gathered on the annotated data. Those counts are shown in the following tables. TABLE II shows the total number of student's requests recorded in the log files, the number of “meaningful” (non garbled, non interrupted, etc.) tutor's answers, grouped by the type of request.

TABLE II. NUMBER OF STUDENT'S REQUESTS AND TUTOR'S ANSWERS.

Consult type	***# of student's requests***	***# of meaningful tutor's answers***
Consult Indicator	72	37
Consult RU	348	235
Consult Debrief	2	1
Show Suspicious RUs	1	1
Total	423	274

From the table above, we can see that the preferred type of student request is *Consult RU*. Most probably this is because students feel they can receive more tips asking for advice about a replaceable unit, rather than asking for an explanation about an indicator reading. However, DIAG thinks the opposite way: it gives much more information answering a *Consult Indicator* query than answering a *Consult RU* query. See Figure 16 for an example.

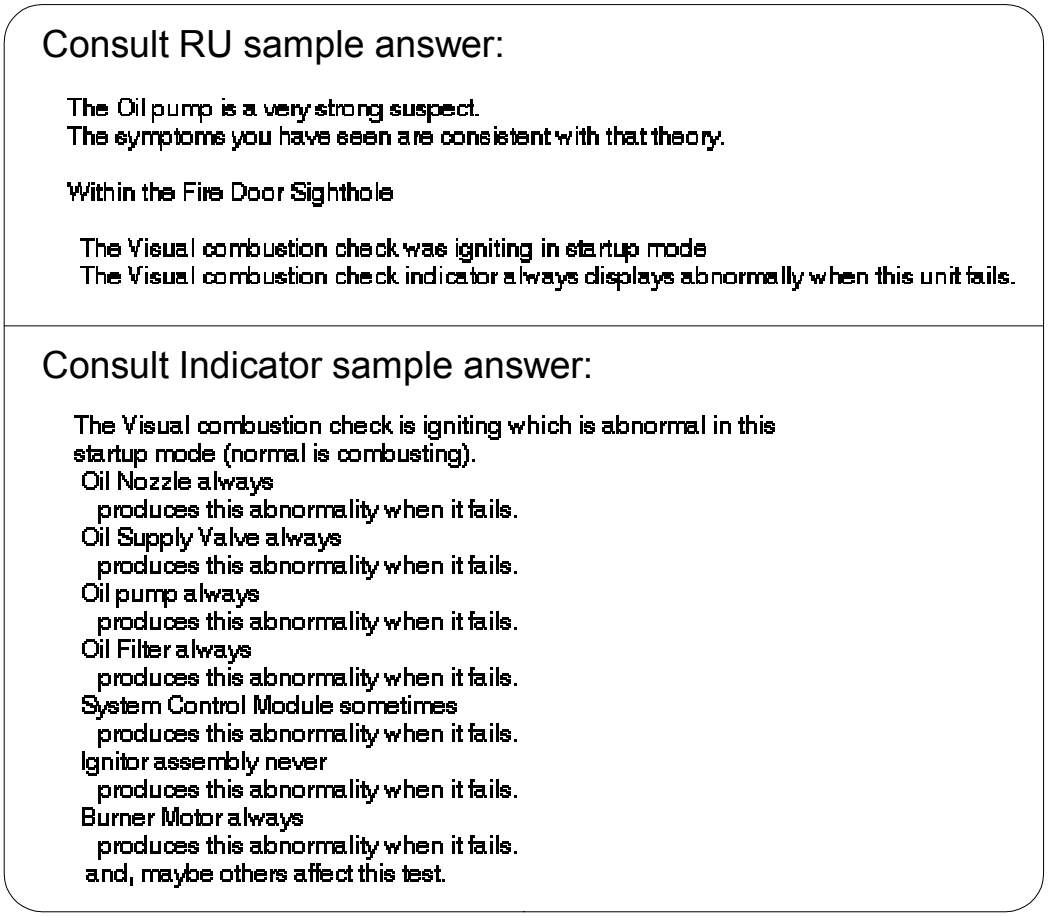

Figure 16. Comparison between DIAG's Consult RU and Consult Indicator answers.

It would have been logical to focus on improving the natural language output of the *Consult RU* queries. However, as stated before, DIAG provides much more information for *Consult Indicator* answers. This means there is much more material for the linguistic planner to work on. So, I chose to concentrate only on the *Consult Indicator* type of queries, manipulating the information DIAG provides in each turn. In the following paragraphs, a discussion of the relationships between actual annotation and system design guidelines will be presented.

4.4 **Improving *Consult Indicator* answers**

Let us have a closer look at the structure of the original DIAG's answer to a *Consult Indicator* query. DIAG provides the following information (look at Figure 16 to see an example):

1) The reading of the indicator to whom the query refers. Example: "The visual combustion check is igniting."
2) The normality or abnormality status of the reading itself. Example: "[...] which is abnormal in this startup mode."
3) If the reading is abnormal, DIAG names also the normal state of the indicator. Example: "[...] (normal is combusting)."
4) Then, DIAG shows a list of sentences with replaceable units as subject. Examples: "Oil nozzle always produces this abnormality when it fails," "Ignitor assembly never produces this abnormality when it fails." What is the meaning of this list? DIAG wants to provide an inferential reasoning that we could summarize as follows: "Indicator K shows an abnormal reading. If the replaceable units X, Y, and Z were broken, then indicator K would show an abnormal reading." Notice that this is not logically equivalent to the statement "Indicator K is abnormal, so replaceable units X, Y, and Z are broken."
5) Finally, the generic advice "maybe others affect this test" is displayed.

In the first version of DIAG-NLP, the information communicated was exactly the same; the difference was in how this information was presented. In DIAG-NLP3 this is no longer true. Some information has been omitted, other information has been modified. The reason of these changes comes from the DIAG corpus analysis.

4.4.1 ***Indication* and *indicator***

We saw that DIAG talks about the state of the indicator being consulted (look at point 1). Do human tutors do it? If so, how? Let us examine the following table, showing the count of the values of the parameters for *indication* and *indicator* tags.

TABLE III. <INDICATION> AND <INDICATOR> TAGGING COUNTS.

Tag name	***Parameter name***	***Parameter meaning***	***Parameter value***	***# of occurrences***
indication				137
	value	It states whether the value of the indicated quantity is present or not.	present	4
			absent	23
			unspecified	110
				137
	qualitative	It indicates if quantitative information is expressed qualitatively.	y	19
			n	1
			unspecified	117
				137
	status	It specifies the status of the indication (normal or abnormal).	normal	50
			abnormal	42
			unspecified	45
				137
	directness	It specifies whether the indication is given in an explicit, implicit, or aggregated (summary) way.	explicit	3
			implicit	2
			summary	0
			unspecified	132
				137
	reality	It states if the indication refers to a real or a hypothetical situation.	real	100
			hypothetical	36
			unspecified	1
				137

Tag name	*Parameter name*	*Parameter meaning*	*Parameter value*	*# of occurrences*
indicator				193
	directness	It specifies whether the object is mentioned directly, indirectly, or in an aggregated way.	explicit	45
			implicit	110
			summary	38
			unspecified	0
				193
	senseref	It marks if there is a general description of the indicator (sense) or its real name (reference).	sense	99
			reference	47
			unspecified	47
				193

The *indication* and *indicator* tags apply to the tutor's answers. They mark where the tutor talks about an indicator object (*indicator* tag) or about the state information coming from an indicator (*indication* tag). An *indication* always entails an *indicator*, so in a correct annotation the number of *indication* marks should be less or equal to the number of *indicator* marks. In our annotated log files, we count 137 *indication* tags and 193 *indicator* tags; this is consistent with the previous constraint.

Notice that 193 indicator tags, over 274 meaningful tutor answers, is a noticeable number. This means that tutors talk about indicators in their answers, so we must use this information in DIAG-NLP3 as well.

Let us have a closer look at *indication* tag parameters. The number of “unspecified” values is very high for all *value*, *qualitative*, and *directness* parameters, so I do not focus my attention on them. About *status* parameter, we can count approximately the same number of instances of the three possible values (“normal”, “abnormal”, “unspecified”); we can deduce that tutors do talk about the indicator, whatever

its status is. The “real” value for *reality* outweighs the “hypothetical” value; this is good news, because DIAG information is about the “real” state of indicators.

The *indicator* tag parameters suggest interesting things. The “implicit” value for *directness* outnumbers the “explicit” value (110 against 45). This means that tutors talk about the object referred to by the indicator, they usually do not talk about the indicator itself; for example, they would not say “the water temperature gauge reading is too low,” they would say “the water is too cold.” The number of “sense” instances for *senseref* is greater than the “reference” ones. I suspect this is related to the fact described before about directness, so it makes it stronger.

4.4.2 ***Operationality* and *ru***

As we did with *indication* and *indicator*, let us examine the counts of *operationality* and *ru* tags.

TABLE IV. <OPERATIONALITY> AND <RU> TAGGING COUNTS.

Tag name	***Parameter name***	***Parameter meaning***	***Parameter value***	***# of occurrences***
operationality				289
	condition	It specifies the status of the replaceable unit (ok or broken).	ok	114
			broken	130
			unspecified	44
				288
	reality	It states if the operationality information refers to a real or hypothetical situation.	real	126
			hypothetical	162
			unspecified	1
				289
ru				551
	directness	It specifies whether the object is mentioned directly, indirectly, or in an aggregated way.	explicit	410
			implicit	40
			summary	101
			unspecified	0
				551
	senseref	It marks if there is a general description of the replaceable unit (sense) or its real name (reference).	sense	16
			reference	409
			unspecified	126
				551

First of all, we can see that the number of *ru* instances is very high, 551 over 274 meaningful tutor turns. Replaceable units are really the core subject of dialogues. Students show interest in learning about them, and tutors talk about them a lot. So, particular care must be taken about replaceable units communication.

The *condition* of the replaceable unit ("ok" or "broken") is told by the tutor, so this is important information to communicate. About the *reality*, we can see that "hypothetical" has quite a large number of instances. In fact, tutors speculate a lot about replaceable units, in order to make students examine various possible situations.

The *directness* parameter is very interesting. Its "explicit" value outnumbers "implicit." This is another proof that human tutors prefer to talk about concrete objects rather than talking about indirect readings. This is coherent with what we noticed in the previous analysis of the *directness* attribute of the *indicator* tag; in the previous case (*indicator)*, a concrete object is made explicit with an implicit reference to the indicator; in the latter case (*ru)*, a concrete object is already explicit, so "explicit" is the preferred reference style used by tutors. So, the duality between these results is coherent. We can see also a meaningful amount of "summary" values for directness. This suggests that tutors perform a lot of aggregation over the replaceable units. This evidence becomes stronger if we consider that not all of the replaceable units in our home heat domain can be aggregated; in fact, only in a few cases functional aggregation is possible. For example, it is difficult to think about a meaningful way to aggregate the "burner motor" and the "water pump," because in our domain the only thing they have in common is the fact that they are both replaceable units. Thus, the high number of "summary" values suggests that tutors perform functional aggregation as much as they can, wherever it is possible.

The large number of "reference" instances for *senseref* is consistent with the dominance of "explicit" in *directness*, under the hypothesis that those two parameters are somewhat related.

4.4.3 ***Included*, *excluded*, and *contradicted***

The first thing to notice looking at the *included*, *excluded*, and *contradicted* annotations is that the number of *excluded* tags outweighs the number of *included* tags. This means that human tutors use only a small portion of DIAG's original information. Unfortunately, it is not easy to infer exactly what pieces of DIAG's information are included explicitly by tutors. This is because the coding scheme used for the annotation allowed a granularity that proved to be too coarse in some cases. For example, some replaceable units have been marked as "included" even if tutors did not talk about them explicitly.

So, I had to inspect the original texts by myself, in order to extract the information I was looking for. The main result I found is that the information of type "sometimes abnormal," "never normal," "never fuzzynormal," "never abnormal," and "no effect" can be ruled out, because its inclusion in the tutor's answers is minimal. This information refers to the likelihood that a replaceable unit is related to the observed malfunction. It was used by DIAG to build point 4 of its answers.

TABLE V. <INCLUDED>, <EXCLUDED>, AND <CONTRADICTED> TAGGING COUNTS.

Tag name	***Parameter name***	***Parameter meaning***	***Parameter value***	***# of occurrences***
included				273
	actual_values	It states whether the actual values are present or not in tutor's discussion.	present	28
			absent	66
			unspecified	0
				94
	normality	It states whether the state of the reading (normal or abnormal) is present or not in tutor's discussion.	present	92
			absent	4
			unspecified	0
				96
	combined	It indicates if the tutor aggregated more elements from DIAG in his answer.	y	25
			n	0
			unspecified	0
				25
	augmented	It indicates if the tutor augmented the information from DIAG in his answer.	y	2
			n	0
			unspecified	0
				2
	suspectability	It indicates whether the tutor included the “suspectability” information in his answer.	y	176
			n	0
			unspecified	0
				176
	rev_susp	It flags DIAG's “review-suspicions” mode.	review_susp	3
			unspecified	0
				3
excluded				486
	rev_susp	It flags DIAG's “review-suspicions” mode.	review_susp	23
			unspecified	0
				23
contradicted				15
	rev_susp	It flags DIAG's “review-suspicions” mode.	review_susp	0
			unspecified	0
				0

4.4.4 ***Probsolving***

A tag that provides another bit of useful information is *probsolving*. Let us see some statistics for this tag.

TABLE VI. <PROBSOLVING> TAGGING COUNTS.

Tag name	***Parameter name***	***Parameter meaning***	***Parameter value***	***# of occurrences***
probsolving				199
	prog_direction	It states whether the advice is about the usage of the tutoring program or not.	y	145
			n	0
			unspecified	54
				199
	domain_direction	It states whether the advice is about the problem to be solved (home heating domain) or not.	y	136
			n	0
			unspecified	63
				199
	other_direction	It specifies if the advice is about something different from the program or the domain.	y	0
			n	0
			unspecified	199
				199

Although these numbers give the impression of an incorrect parameter filling (there are no instances of "n" at all and there are too many "unspecified" instances), we notice that the total number of probsolving tags is very high. In fact, tutors almost always address the students directly, giving him/her advice; sentences like "check this...", "test that...", "make sure that..." are very common. This is made evident by the high number of "y" in the *domain_direction* parameter.

The high number of "y" in the *prog_direction* (e.g., in sentences like "click on the Consult button"), shows that tutors have to spend time telling the students how to use the program, instead of

focusing on the problem to be solved. This amount of directions could probably be reduced by designing a more intuitive Graphical User Interface.

The *other_direction* parameter should not be taken into account, because it has “unspecified” instances only.

4.5 **Summary**

A lot of features of human tutors' answers have been discovered from the analysis of the annotated data. The most important patterns are summarized below.

1) Tutors usually talk about the object the indicator refers to, not about the indicator itself. This is made evident by *indication* and *indicator* markings.
2) Tutors talk a lot about replaceable units, they mention their status, and they perform significant aggregation on them. These features can be inferred looking at the *operationality* and *ru* markings.
3) Tutors rule out a large amount of information from the original DIAG's *Consult Indicator* answers. This is shown by *included*, *excluded*, and *contradicted* markings.
4) Tutors organize many sentences in the form of direct advice to the student. This results from the *probsolving* marking.

The next Chapter shows how these features have been implemented in DIAG-NLP3's Sentence Planner.

5. SENTENCE PLANNER IMPLEMENTATION

This Chapter shows how I implemented the features highlighted in Chapter 6. The implementation was relatively easy, because both the Language Generation Engine and the Knowledge Management Module, that will be discussed in Chapter 7 and Chapter 8, proved to fit their tasks very well.

As stated before, only the answers to *Consult Indicator* queries are processed by DIAG-NLP3. All the other queries are still processed by the first NLP-enhanced version of the system, DIAG-NLP1. Figure 17 shows how the different kind of queries are processed by the system.

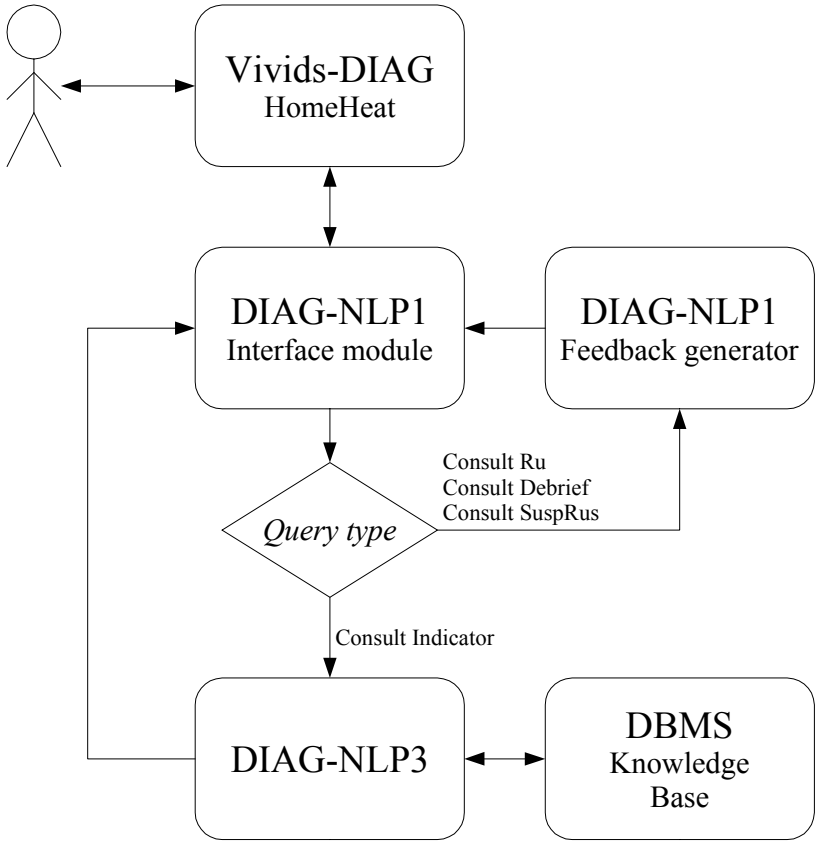

Figure 17. Query routing in DIAG-NLP3.

5.1 **Sentence Planner structure**

The decision process used by the new Consult Indicator Sentence Planner is depicted in Figure 18.

First of all, the system says something about the indicator reading. Some processing is performed here to make the indicated object explicit. For example, given DIAG's information “Indicator name=water_temperature_gauge state=50 normalState=100”, DIAG-NLP3 would say “The water is too cold.”

Then, the system talks about the replaceable units that could be related to the indicator state. This is done only if the indicator reading is abnormal; otherwise nothing more is said.

A lot of processing is performed over the replaceable units list provided by DIAG.

1) Irrelevant units are discarded; a unit is considered irrelevant if it does not affect the normality of the indicator reading when it is broken.
2) Units are partitioned in subsets according to their location. A new sentence will be created for each subset.
3) Functional aggregation is performed on each subset of units.
4) Sentences are organised in a “direct advice” form, i.e., imperative, like “check the system control module.”

Points 1 and 3 are described in more detail in the following sections.

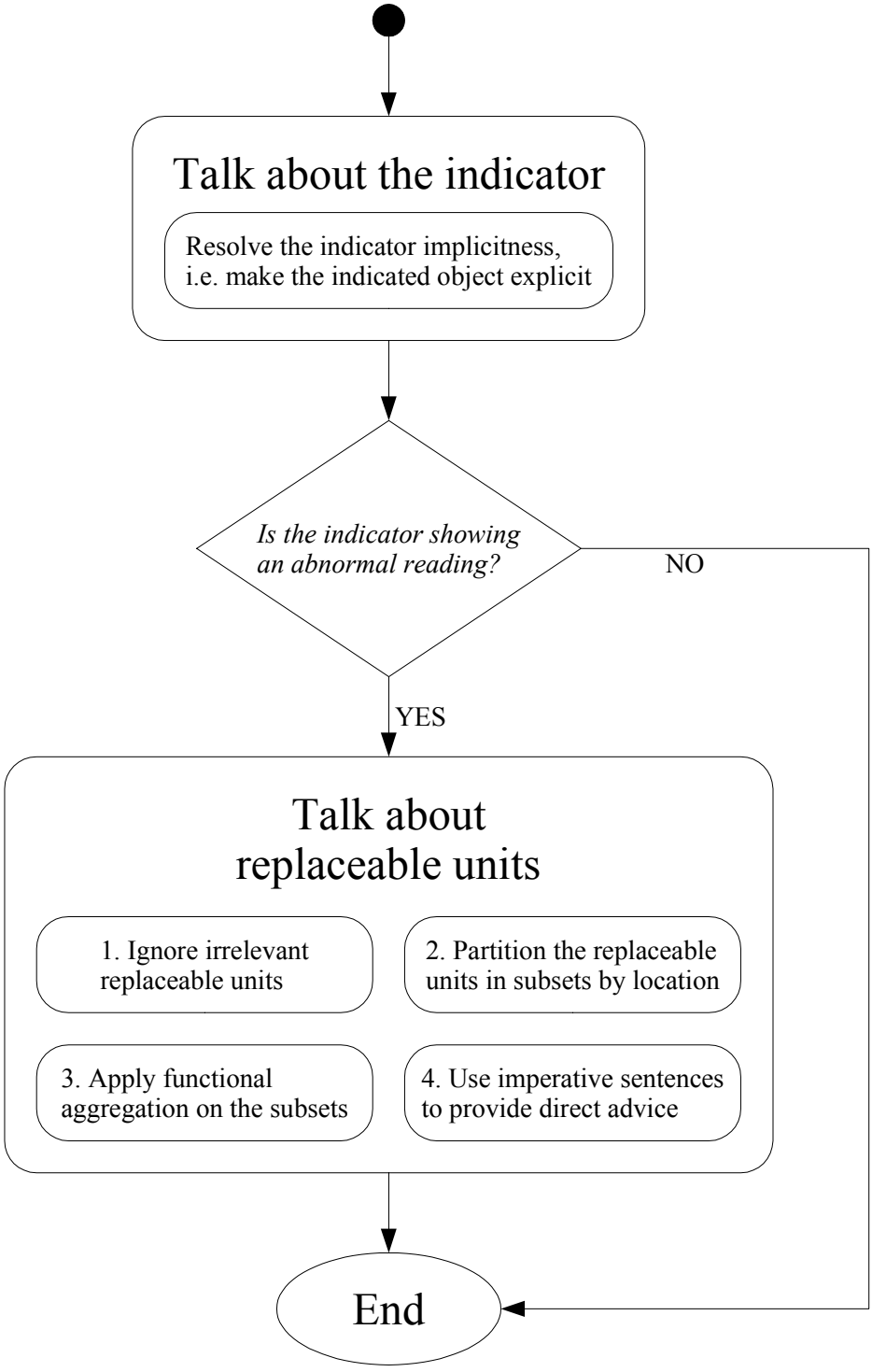

Figure 18. DIAG-NLP3's feedback sentence planner.

5.2 **Implicit indication resolution**

The *implicit indication resolution* is easily done by using the *relation mechanis*m. This will be described in more detail in Chapter 7. Briefly, a *relation* is a tuple of named attributes attached to an *object*. So, relations are used to express properties about objects in our domain. A collection of relations is stored in a Knowledge Base and they are retrieved at run-time when needed. The Knowledge Base implemented in DIAG-NLP3 will also be discussed in Chapter 7.

Each indicator object has a relation called *implicitness* (see TABLE X in Chapter 7). The Sentence Planner retrieves the needed instance of this relation from our Knowledge Base and makes its value the new logical subject of the sentence. For example, consider "water_temperature_gauge." The value of the parameter *reference* of the relation *implicitness* associated to this object is "water." The system will consider "water" the new subject, instead of "water_temperature_gauge." Then, the system will retrieve the *normality* relation instance of the new subject. In our example, it will retrieve the *normality* of "water" instead of the *normality* of "water_temperature_gauge."

The system deducts the status of the indicator from DIAG's information, for example it knows the reading is in a state of *abnormality2* (the reading is lower than it should be). So, DIAG-NLP3 will use the *abnormality2* value of the *normality* relation associated to "water" as the logical direct object of its sentence, in our case "too cold." The main verb will be, of course, "to be," because we are talking about a state.

At this point, the system has a subject, a verb, and a direct object, so it can build a complete sentence using its Generation Engine (Chapter 6).

5.3 **Functional aggregation**

Functional aggregation is used to replace long lists of objects with a single one representing some shared properties of the entire list. For example “oil filter,” “oil nozzle,” “oil pump,” and “oil supply valve” become “units along the path of the oil.”

To implement it, I used a simple relation (see Chapter 7), called *functional_aggregation*. This relation has two parameters: *aggregator* and *cardinality*. *Aggregator* contains the name of an object that is representative of the entire set; *cardinality* indicates how many objects compose the aggregator. Notice that we are dealing with bounded functional aggregation. For example,

```
functional_aggregation(“oil filter”,
   aggregator = “units along the path of the oil”, cardinality = “4”);
```

means that “oil filter” is one of the four components of the aggregate object “units along the path of the oil.”

The indication of the cardinality is used in this simple heuristic: if all of the components appear in the input list, then the system replaces them with the aggregator; if there are less than one half of the components in the input list, it does not replace them at all; if there are at least one half, but not all, of the components, it replaces them with the aggregator, limited by the quantifier “some of.” For example, a list composed by “oil filter,” “oil supply valve,” and “oil pump” would be replaced with “some of the units along the path of the oil.”

5.4 **DIAG-NLP3's sample output**

Figure 19 and Figure 20 show two examples of DIAG-NLP3's answers to Consult Indicator queries. Notice that the output is much more concise than in the previous versions of the system; this is due to the high amount of aggregation performed and the choice of excluding some of the original information.

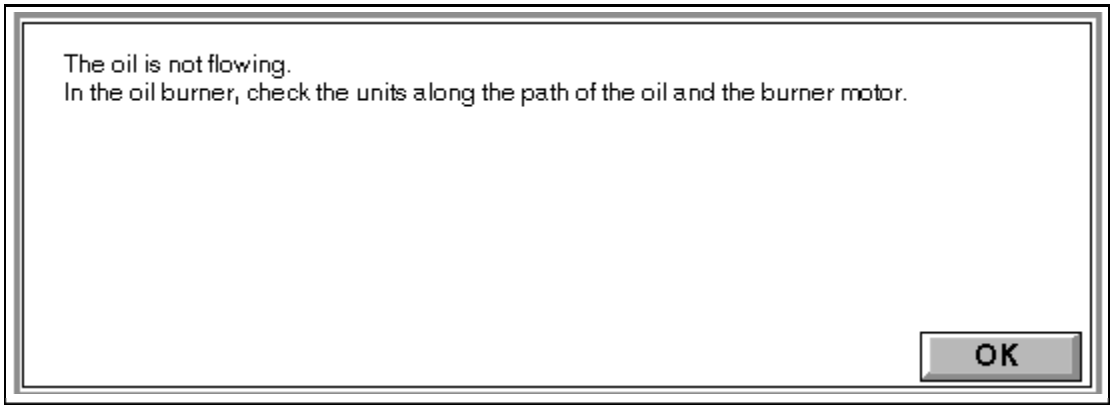

Figure 19. DIAG-NLP3's answer to a Consult Indicator query.

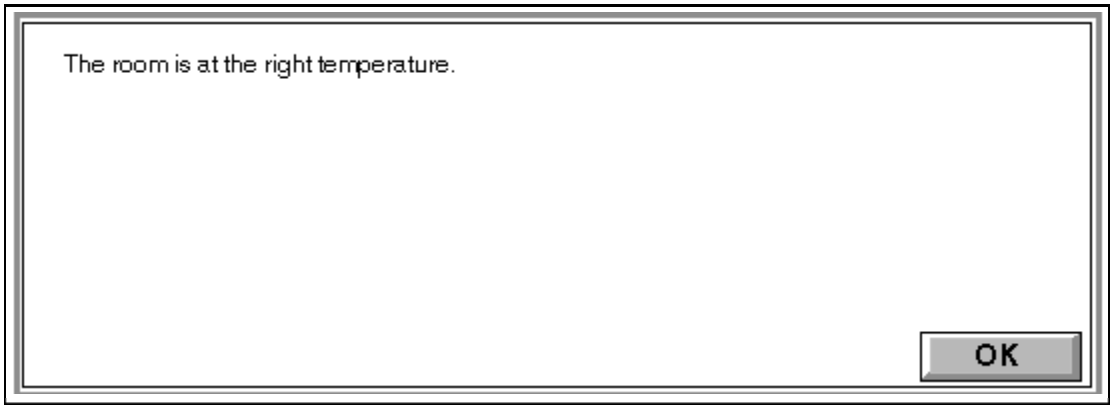

Figure 20. DIAG-NLP3's answer to another Consult Indicator query.

6. LANGUAGE GENERATION ENGINE

6.1 The generation engine in DIAG-NLP1 and DIAG-NLP2

In both DIAG-NLP1 and DIAG-NLP2, the language generation engine is entirely based on EXEMPLARS (White and Caldwell, 1998), a system implemented by CoGenTex Inc. EXEMPLARS is basically a sophisticated template filler. It allows building complex sentence structures using recursion and context-dependent rules. EXEMPLARS works using a set of user-defined rules, called *exemplars*. These *exemplars* represent how the generation process should proceed, driven by the context. EXEMPLARS builds sentences using a tree-like structure, where each node comes from expansion or refinement of a previously generated one. The leaves of this tree represent the generated sentence. EXEMPLARS can be used either to generate full sentences or just to define sentence structures, postponing the surface realization to a later stage of processing.

In DIAG-NLP1 and DIAG-NLP2, the EXEMPLARS system was used to build complete sentences. To do that, the leaves of the tree contain strings of text. This approach is effective but reveals a major limitation: the *exemplars* used are strongly bounded to the specific application and communicative context. An example is shown if Figure 21. The task of this sample *exemplar* is to aggregate the replaceable units by their location. Notice that the actual names of the replaceable units and their locations are hard coded, as well as some fragments of the text to be generated. With this approach, code reuse is practically impossible. Also maintenance is very difficult; to add new features or to modify an existing one, the designer must be aware of the entire *exemplars* structure, because even a little change in one *exemplar* can have strong side effects on the entire *exemplars* system.

As usual in computer science, layering is a key solution to this problem. I decided to redesign the entire language generation subsystem in a layered fashion from scratch.

```
exemplar AggByContainerCode(Vector lists, int levels,
Vector params, String postPhrase) extends AggByType
{


 void sort(Vector list)
 {
  oilBurner = new Vector();
  waterPump = new Vector();
  livingRm = new Vector();
  furnaceSys = new Vector();
  fdsh = new Vector();
  others = new Vector();

  oilBurner.addElement(oilBurnerPreListPhrase);
  waterPump.addElement(waterPumpPreListPhrase);
  livingRm.addElement(livingRmPreListPhrase);
  furnaceSys.addElement(furnaceSysPreListPhrase);
  fdsh.addElement(fireDoorSightHolePrePhrase);
  others.addElement(othersPreListPhrase);

  for (int i = 0; i < list.size(); i++)
  {
   Part part = ((Part)list.elementAt(i));

   SubAssembly container = part.getContainer();


   if (container != null)
   {
    String contName = container.getName();


    if (contName.equals("Oil Burner"))
      oilBurner.addElement(part);
    else
    if (contName.equals("Water Pump & Safety Cutoff
Valve"))
      waterPump.addElement(part);
    else
    if (contName.equals("Living Room"))
     livingRm.addElement(part);
    else
    if (contName.equals("Furnace System"))
     furnaceSys.addElement(part);
    else
     if (contName.equals("Fire Door Sight Hole"))
      fdsh.addElement(part);
    }
    else
    {
     others.addElement(part);
    }
   }
  }

  Vector addNestedLists()
  {
   Vector newVect = new Vector();
   if (oilBurner.size() > 1)
     newVect.addElement(oilBurner);
   if (waterPump.size() > 1)
     newVect.addElement(waterPump);
   if (livingRm.size() > 1)
     newVect.addElement(livingRm);
   if (furnaceSys.size() > 1)
     newVect.addElement(furnaceSys);
   if (fdsh.size() > 1)
     newVect.addElement(fdsh);
   if (others.size() > 1)
     newVect.addElement(others);
   return newVect;
  }

  // Declare a bucket for each container type involved
  String oilBurnerPreListPhrase = "Within the Oil Burner";
  String waterPumpPreListPhrase = "Within the Water
Pump & Safety Cutoff Valve";
  String livingRmPreListPhrase = "Within Living Room";
  String furnaceSysPreListPhrase = "Within the Furnace
System";
  String fireDoorSightHolePrePhrase = "Within the Fire
Door Sighthole";
     String othersPreListPhrase = "others";
     Vector oilBurner;
     Vector waterPump;
     Vector livingRm;
     Vector furnaceSys;
     Vector fdsh;
     Vector others;
    }
```

Figure 21. Sample exemplar used in DIAG-NLP1.

6.2 **Goals**

The three main goals for this part of the system are *ease of use*, *versatility*, and *reusability*.

To improve ease of use, I decided to raise the abstraction level of the interface between the application system (in our case, the system that decides what to communicate) and the generation engine. I wanted sentences that could be programmed by simply filling high-level data structures, without being concerned with some of the low-level details of the grammar.

This increased abstraction level also improves the versatility of the generation system, because the same mechanisms can be used to implement a wide variety of communicative goals, without having to implement new generation code.

Reusability should be granted, removing the domain dependency between the generation system and the application that uses it.

6.3 **Structure of the generation engine**

The structure of the generation engine is composed by three software layers: the *sentence structure definition and validation* layer, the *sentence structure renderer* layer, and the *surface realization layer*. This structure is represented in Figure 22.

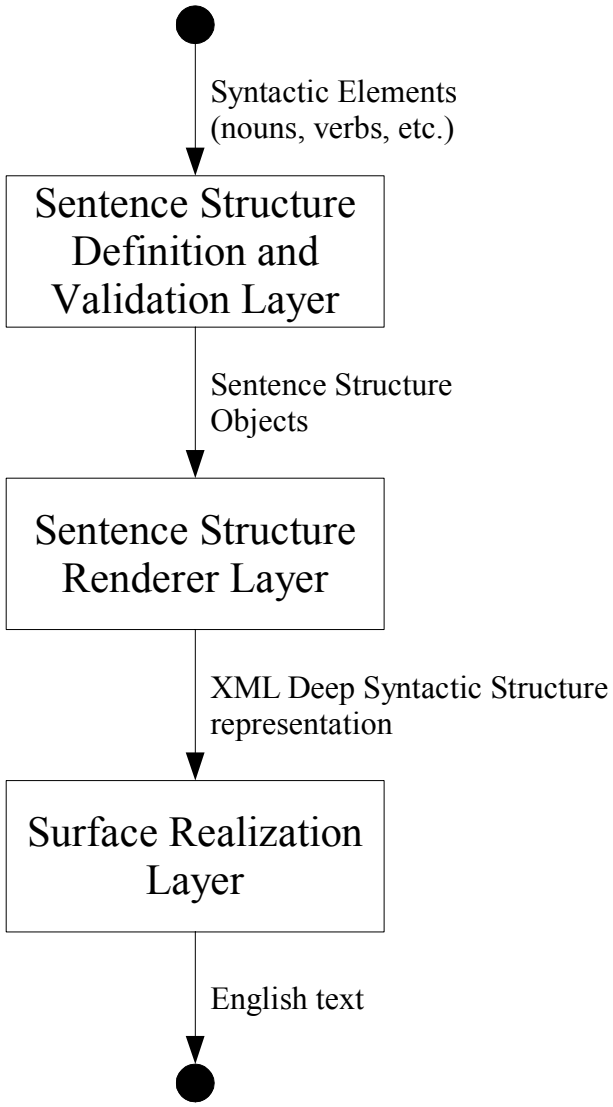

Figure 22. Architecture of DIAG-NLP3's generation engine.

6.3.1 **Sentence structure definition and validation layer**

This is the layer at the highest abstraction level in the generation chain. In this layer, a sentence is defined by filling a Java data structure containing the elements to communicate. Designing the format of this data structure, I had to choose a trade-off between flexibility and ease of use of the structure itself. In fact, a high flexibility in the structure would imply the ability to represent a high variety of complex sentences, at the price of complexity of use. On the other end, a simple and rigid structure would be very easy to use, but would also be very limited in its expressive power. The solution I choose is that of having a quite simple structure, where the composing elements are somewhat flexible and customizable.

A *SyntacticElement* object is a word enriched with its grammatical properties. It can be, for example, a noun, an adjective, or a verb, with the indication of its gender, number, and the other properties needed to identify it in its context. A *SyntacticElement* can have a coordinate or an attribute (a subordinate), allowing the definition of complex coordination and subordination relationships between words.

A *Sentence Structure* is an object composed by six *SyntacticElement* objects: *subject*, *object*, *verb*, *subjectGlobalAttribute*, *preVerbal*, and *sentFinal*. Its *subject* field represents the logic subject of the sentence. Its *objec*t field represents the logic direct object of the sentence. Its *verb* field must be the primary verb of the sentence; *subjectGlobalAttribute*, *preVerbal*, and *sentFinal* can be filled with attributes or indirect object terms.

This kind of structure, in conjunction with the coordination and subordination capabilities of the *SyntacticElement* objects, allows the representation of sentences of medium complexity. It is likely that this degree of complexity would be adequate for the output of our Intelligent Tutoring System.

To define a sentence, the application program will create an empty *SentenceStructure* object and fill it with all the *SyntacticElements* needed. An important property of this mechanism is that *the structure can be filled without having to take care of the filling order*. This is a very important feature that improves the ease of use of the generation engine.

6.3.2 **Sentence structure renderer layer**

This layer converts a *SentenceStructure* object into a Deep Syntactic Structure, a formalism based on Meaning-Text Theory (Mel'cuk et al., 1987; Mel'cuk, 1988). In such a representation, the sentence is defined by a lexicalized tree, where each node is labeled with a lexeme (uninflected word). Such a tree represents the syntactic relationships between the component words, leading to a precise formal representation of the sentence itself. The Deep Syntactic Structure representation is required by the surface generation system *RealPro*, by CoGenTex Inc (Lavoie and Rambow 1997). An example of Deep Syntactic Structure representation in XML format is shown in Figure 23. It represents the sentence "The system control module in the furnace system is a replaceable unit."

```
<?xml version="1.0" encoding="UTF-8"?>
<html>

<dsynts>
  <dsyntnode class="verb" lexeme="be" rel="nil">
    <dsyntnode class="common_noun" lexeme="system_control_module" rel="I"/>
    <dsyntnode class="common_noun" lexeme="replaceable_unit" article="indef" rel="II"/>
    <dsyntnode class="preposition" lexeme="in" position="pre-verbal" rel="ATTR">
      <dsyntnode class="common_noun" lexeme="furnace_system" rel="II"/>
    </dsyntnode>
  </dsyntnode>
</dsynts>

</html>
```

Figure 23. Example of Deep Syntactic Structure in XML format.

The Sentence Structure Renderer Layer has been written using the *Exemplars* framework, by CoGenTex Inc. There is an important difference between how I used *Exemplars* and how the developers of DIAG-NLP1 and DIAG-NLP2 used it. In the previous systems, *Exemplars* was used to carry out the entire generation process, from the beginning to the final sentence. In my system, *Exemplars* performs only an intermediate translation from a high level structure to a more detailed and formalized one.

To build this tree, the system starts generating a node from the main verb, then it recursively links all the other elements of the sentence, making the syntactic relationships between the elements explicit and filling in other details where needed. The output of this processing is an XML tree with the Deep Syntactic Structure representation of the sentence.

Anyway, the user of the generation engine does not need to be concerned about the details of this intermediate stage of the generation process; it is performed automatically by the system when it is needed.

6.3.3 **Surface realization layer**

This layer performs the final stage of the generation process. It takes a Deep Syntactic Structure representation of a sentence and transforms it in a plain text English sentence. I used *RealPro*, by CogenTex Inc., to perform this task (Lavoie et Rambow, 1997). *RealPro* was preferred to other surface realizers, such as the very flexible FUF/SURGE (Elhadad 1993; Elhadad and Robins 1996), because of the steep learning curve of the latter one. *RealPro* provides a native Java Application Program Interface, so it has been relatively straightforward to interface it with the rest of the system. *RealPro* is not a framework (like *Exemplars*), but it is a stand-alone program. The current release of *RealPro* is adequately robust with respect to the specifications, although it has some bugs and oddities.

It is important that the user of my generation engine knows the *RealPro* syntax, because all the grammatical properties of the high-level *SyntacticElement* objects are expressed with the same syntax of RealPro; for example, *sg* stands for number singular, *fem* stands for gender female, and so on. I chose that syntax for simplicity, since there was no reason to define a new syntax from scratch. For a complete reference of the attributes of the implemented subclasses of S*yntacticElement*, see Appendix B.

Notice that the current implementation does not entirely exploit the expressive power of *RealPro*. It is possible to realize more grammatical features simply by implementing new subclasses of the *SyntacticElement* class, according to RealPro syntax.

6.3.4 **Tutorial**

Let us see how this works in practice (note: this is a technical tutorial, the reader may skip it if he/she is not interested in the implementation details). After installing the system, consisting in a few Java packages, a programmer can use it simply by invoking some Java methods.

Here follows an example. It is not about the Home Heating domain, in order to show that this Generation Engine can be used also in completely different contexts. In our example, we want to build a sentence with these elements:

1) the main action is eating;
2) the actors are kids;
3) the actors are young;
4) the object of the action is an ice-cream;
5) the object has two features, it is fresh and tasty;
6) the action takes place in the afternoon.

We would expect a sentence like “Young kids eat a fresh and tasty ice cream in the afternoon”.

All the classes we need are stored in the packages diagnlp3.sel and diagnlp3.real.

```
import diagnlp3.sel.*;
import diagnlp3.real.*;
```

The first thing we need to do is to create an empty SentenceStructure object.

```
SentenceStructure ss = new SentenceStructure();
```

Now we create the *SyntacticElement* objects we need. We have to use the right *SyntacticElement* subclasses, in this case *Verb*, *CommonNoun*, and *Adjective*.

```
Verb action =  new Verb("eat");
CommonNoun actor1 =  new CommonNoun("kid");
CommonNoun object1 = new CommonNoun("ice_cream");
Adjective attribute1 = new Adjective("young");
Adjective attribute2 = new Adjective("fresh");
Adjective attribute3 = new Adjective("tasty");
CommonNoun when = new CommonNoun("afternoon");
```

Now we want to relate the term *attribute1* to the term *actor1*, to state that the kids are young.

```
actor1.setAttribute(attribute1);
```

Now we have to specify some additional properties of actor1, such as the number (plural) and the article (indefinite). It is necessary to specify those details to override the default assumption of number singular and article definite.

```
actor1.number = "pl";
actor1.article = "indef";
```

Now we want to relate the coordination of *attribute2* and *attribute3* (“fresh and tasty”) to *object1*., and we specify that the article of *object1* is indefinite.

```
object1.setAttribute(attribute2);
attribute2.setCoordinate(attribute3);
object1.article = "indef";
```

To make the term *when* a time object, we specify the temporal preposition “in”.

```
when.setPreposition("in");
```

Now all the elements are set up. We can fill in the SentenceStructure we created before.

```
ss.verb = action;
ss.subject = actor1;
ss.object = object1;
ss.sentFinal = when;
```

Before we proceed with the generation process, we need to validate the structure we filled in. This will automatically fix some agreement problems and other details, so that the programmer does not have to take care of them.

```
ss.validateStructure();
```

Now we can carry on the generation process. First, we need to create a *SentenceBuilder* object.

```
SentenceBuilder builder = new SentenceBuilder();
```

Then, we need an instance of a *SurfaceRealizer*. Since the initialization of a SurfaceRealizer takes some time, it might be a good idea to create such an object at the very beginning of a program, and to reuse it for all the generation tasks we need.

```
SurfaceRealizer realizer = new SurfaceRealizer();
```

Finally, we can issue the generation command. It will return a String with the complete sentence.

```
String sentence;
sentence = builder.renderSentenceStructure(ss, realizer);
```

We can print out the result now.

```
System.out.println(sentence);
```

This will be the output of the last command:

```
Young kids eat a fresh and tasty ice cream in the afternoon.
```

The generation process we explored in this tutorial might appear complicated, but it is not. In fact, this generation engine is suitable for automated processing, because it allows the computation to be split in different stages, and most of them can be performed out of order. Although a lot of details must be specified, this approach allows the discourse to be designed at a pretty high level of abstraction.

7. DOMAIN KNOWLEDGE REPRESENTATION

7.1 The need for an external Knowledge Base

Since its very beginning, the development of a Natural Language Interface for Vivids-DIAG had to face a sad fact: the internal knowledge representation of the domain (in our case, the home heating world) is hidden in the DIAG code and it is completely inaccessible from an external program. That means it became necessary to code some kind of "external" knowledge representation to allow the NLG add-on to make inferences about the world it is talking about.

In DIAG-NLP1, this knowledge base was mixed up with the generation engine. There were some Java classes representing all the objects in the domain, and they were used in the generation process. The problem of this approach is that it makes it impossible to extend or modify the domain without being concerned with the details of the generation process.

In DIAG-NLP2, a frame-based knowledge representation was added. It was built using the SnePS representation system (Shapiro and Rapaport, 1992). This type of knowledge representation is easier to extend because it is defined independently of the generation engine. For a complete description, see (Haller and Di Eugenio, 2002).

I decided to do something different in DIAG-NLP3. My goal was to build a knowledge representation that was simple, easy to read, easy to maintain, yet powerful enough to perform some automated reasoning.

I explored the option of using CLASSIC (Brachman et al., 1990). CLASSIC is a very interesting system, and it allows the representation of complex ontologies in an easy way. However, I decided not to use it for several reasons. First of all, CLASSIC is difficult to interface and query. It is built in LISP, so

it would have required the usage of a Java-LISP bridge, because our system is written in Java. In addition, its inference power would have been unnecessary for our needs.

Thus, I decided to represent the knowledge base with a standard relational database. This has many advantages: database technology is widespread, easy to use, reliable, fast, and, above all, has a standard and well-known query language (SQL). It is straightforward to access a relational database from a Java program, using a standard interface (JDBC).

7.2 **Structure of the knowledge representation system**

The knowledge representation system is composed of two parts: a set of tables, stored in a DBMS, and a set of Java classes. At run time, the system retrieves the necessary information from the tables and stores it into suitable Java structures for further processing.

In my Java representation, the main unit of knowledge is called a *relation*. I implemented it in a class called *Relation*. Relations are tuples of named attributes attached to an *object*. The word object here must not be confused with the concept of object in a programming language. An object is intended as the name of a "real" object in our ontology. These are examples of relations:

```
place("burner motor", location = "oil burner");
is_a("burner motor", parent = "replaceable unit");
normality("burner motor", normality = "ok",
          abnormality1 = "spinning too fast",
          abnormality2 = "spinning too slow");
```

In the previous examples, those three relations stated three properties of the object "burner motor" that could be used to make inferences about the burner motor itself. The main idea is that sentences can be planned statically using the general relations, and then rendered at run-time using the data collected from the database.

Let us see an example. Suppose that we want to state something about an object's place and "parent" (i.e., "The burner motor in the oil burner is a replaceable unit"). We could plan a sentence like this:

```
(object) in (place:location) [to be] (is_a:parent)
```

Once we know the object we are talking about, it is straightforward to retrieve its properties and provide the right parameters to the generation engine.

7.3 **Database table structure**

The primary table in our Knowledge Base is called *relations*. This table is used to describe the structure of all the relations in the database. It is mandatory for the correct data retrieval by the Java interface. This is the table definition:

```
relations (name, type, param_number, param_names);
```

- *name* is the name of the relation;
- *type* is a general attribute of the relation. This information is currently unused in DIAG-NLP3;
- *param_number* defines the number of parameters (attributes) of the relation;
- *param_names* describes the names of each parameter of the relation, separated by commas.

The actual content of this table in our Home Heat KB is shown in TABLE VII.

TABLE VII. "RELATIONS" TABLE IN THE HOME HEAT KB.

name	***type***	***param_number***	***param_names***
is_a	static	1	parent
place	static	1	location
implicitness	static	1	reference
normality	static	3	normality, abnormality1, abnormality2
functional_aggregation	static	2	aggregator, cardinality

Each relation described in the *relations* table has been implemented in a specific table, with the name of the relation itself.

7.3.1 ***Is_a* relation**

The *is_a* relation models a simple hierarchical ontology. It relates an object with a "parent" of the object itself (e.g., the "furnace system" is a "replaceable unit").

```
is_a (object, parent);
```

TABLE VIII shows the actual content of this table.

TABLE VIII. "IS_A" TABLE IN THE HOME HEAT KB.

object	***parent***
burner_motor	replaceable_unit
burner_motor_rpm_gauge	indicator
fire_door_and_sight_hole	place
furnace_system	place
ignitor_assembly	replaceable_unit
indicator	thing
living_room	place
oil_burner	place
oil_filter	replaceable_unit
oil_flow_indicator	indicator
oil_nozzle	replaceable_unit
oil_pump	replaceable_unit
room_temperature_gauge	indicator
oil_supply_valve	replaceable_unit
photoelectric_cell	replaceable_unit
place	thing
replaceable_unit	thing
water_temperature_safety_cutoff_valve	replaceable_unit
system_control_module	replaceable_unit
transformer	replaceable_unit
visual_check_of_oil_filter	indicator

object	*parent*
visual_combustion_check	indicator
water_pump	replaceable_unit
water_pump_sound	indicator
water_temperature_gauge	indicator
ir_sensor	replaceable_unit
room_water_supply_valve	replaceable_unit
room_occupancy_test_point	indicator
oil_supply_valve_signal	indicator
room_water_valve_signal	indicator
burner_motor_signal	indicator
water_pump_signal	indicator
ignition_signal	indicator
water_temperature_test_point	indicator
desired_temp_test_point	indicator
current_temp_test_point	indicator
combustion_test_point	indicator

7.3.2 ***Place* relation**

The *place* relation defines a spatial location for an object. For example, the “oil filter” is in the “oil burner;” the “oil burner” is in the “furnace system.”

```
place (object, location);
```

Look at TABLE IX for the actual content of this table.

TABLE IX. “PLACE” TABLE IN THE HOME HEAT KB.

object	***location***
burner_motor	oil_burner
fire_door_and_sight_hole	furnace_system
ignitor_assembly	oil_burner
oil_burner	furnace_system
oil_filter	oil_burner
oil_nozzle	oil_burner
oil_pump	oil_burner
oil_supply_valve	oil_burner
photoelectric_cell	oil_burner
water_temperature_safety_cutoff_valve	furnace_system
system_control_module	furnace_system
transformer	oil_burner
water_pump	furnace_system
ir_sensor	living_room
room_water_supply_valve	living_room
room_occupancy_test_point	system_control_module
oil_supply_valve_signal	system_control_module
room_water_valve_signal	system_control_module
burner_motor_signal	system_control_module
water_pump_signal	system_control_module
ignition_signal	system_control_module
water_temperature_test_point	system_control_module
desired_temp_test_point	system_control_module
current_temp_test_point	system_control_module
combustion_test_point	system_control_module

7.3.3 ***Implicitness*** **relation**

The *implicitness* relation is used to model a change of point of view in the description of a fact.

```
implicitness (object, reference);
```

TABLE X shows the actual values in this table.

TABLE X. "IMPLICITNESS" TABLE IN THE HOME HEAT KB.

object	***reference***
room_temperature_gauge	room
burner_motor_rpm_gauge	burner_motor
water_temperature_gauge	water
water_pump_sound	water_pump
visual_combustion_check	combustion
visual_check_of_oil_filter	oil_filter
oil_flow_indicator	oil
room_occupancy_test_point	room_occupancy_test_point
oil_supply_valve_signal	oil_supply_valve_signal
room_water_valve_signal	room_water_valve_signal
burner_motor_signal	burner_motor_signal
water_pump_signal	water_pump_signal
ignition_signal	ignition_signal
water_temperature_test_point	water_temperature_test_point
desired_temp_test_point	desired_temperature_test_point
current_temp_test_point	current_temperature_test_point
combustion_test_point	combustion_test_point

7.3.4 ***Normality*** **relation**

The *normality* relation is used to describe the possible "working" and "broken" states of an object. For example, "water" can be "at the right temperature," "too hot," or "too cold."

```
normality (object, normality, abnormality1, abnormality2);
```

The actual values stored in this table are shown in TABLE XI.

TABLE XI. "NORMALITY" TABLE IN THE HOME HEAT KB.

object	***normality***	***abnormality1***	***abnormality2***
room	at_the_right_temperature	too_warm	too_cold
burner_motor	ok	spinning_too_fast	spinning_too_slow
water	at_the_right_temperature	too_hot	too_cold
water_pump	ok	broken	
combustion	normal	abnormal	
oil_filter	clean	clogged	
oil	flowing	not_flowing	
room_occupancy_test_point	normal	too_high	too_low
oil_supply_valve_signal	normal	too_high	too_low
room_water_valve_signal	normal	too_high	too_low
burner_motor_signal	normal	too_high	too_low
water_pump_signal	normal	too_high	too_low
ignition_signal	normal	too_high	too_low
water_temperature_test_point	normal	too_high	too_low
desired_temperature_test_point	normal	too_high	too_low
current_temperature_test_point	normal	too_high	too_low
combustion_test_point	normal	too_high	too_low

7.3.5 ***Functional_aggregation*** **relation**

The *functional_aggregation* relation is used to perform functional aggregation on the objects.

```
functional_aggregation (object, aggregator, cardinality);
```

TABLE XII shows the actual values of this table.

TABLE XII. "FUNCTIONAL_AGGREGATION" TABLE IN THE HOME HEAT KB.

object	***aggregator***	***cardinality***
oil_filter	units_along_the_path_of_the_oil	4
oil_nozzle	units_along_the_path_of_the_oil	4
oil_pump	units_along_the_path_of_the_oil	4
oil_supply_valve	units_along_the_path_of_the_oil	4
ignitor_assembly	electrical_devices	4
photoelectric_cell	electrical_devices	4
transformer	electrical_devices	4
burner_motor	electrical_devices	4
room_occupancy_test_point	signals_to_the_scm	10
oil_supply_valve_signal	signals_to_the_scm	10
room_water_valve_signal	signals_to_the_scm	10
burner_motor_signal	signals_to_the_scm	10
water_pump_signal	signals_to_the_scm	10
ignition_signal	signals_to_the_scm	10
water_temperature_test_point	signals_to_the_scm	10
desired_temp_test_point	signals_to_the_scm	10
current_temp_test_point	signals_to_the_scm	10
combustion_test_point	signals_to_the_scm	10

7.4 **Tutorial**

This is a technical tutorial; readers may skip it if they are not interested in these details. Suppose we have a few strings, containing the names of some objects. We want to use our Knowledge Base to retrieve information about those objects. Let us call those strings *object1*, *object2*, and *object3*. For example we might have:

```
String object1 = "oil_filter";
String object2 = "water_pump";
String object3 = "ir_sensor";
```

The classes that allow us to manage the KB are in package *diagnlp3.agg*. We must include it.

```
import diagnlp3.agg.*;
```

The first step is to create a bridge between our Java program and our Relational Database. This bridge is provided by the class StaticKnowledgeBase.

```
StaticKnowledgeBase kb = new StaticKnowledgeBase();
```

Notice that no parameter is required by the constructor. The parameters of the database, like its name, address, and password, are defined in the class *diagnlp3.misc.Constants*. This might appear a bad programming style, but it has been chosen to shield the Application Program Interface of the Knowledge Base from the underlying database structure. We must be able to change the database, or even to change the representation itself with a different technology (e.g., CLASSIC) without forcing the user to be concerned with the change.

After the creation of this object, we are ready to submit queries to our KB. Suppose we want to know where our three objects are.

```
Relation place1 = kb.getSingleRelation(object1, "place");
Relation place2 = kb.getSingleRelation(object2, "place");
Relation place3 = kb.getSingleRelation(object3, "place");
```

That's it! If the relation "place" was found for our three objects, then the three variables *place1*, *place2*, and *place3* would contain three new *Relation* objects storing the required information. If such information was not available for one or more objects, the variables would have been set to *null*.

Using a *Relation* object is easy. It has methods to retrieve its attribute either by name or by index. In our previous example, we could retrieve a string from the relation *place1* is these two equivalent ways:

```
String p1a = place1.getValue("location");
String p1b = place1.getValue(1);
```

Since a *Relation* can have multiple attributes, the mechanism described before makes it easy to manage that.

8. EVALUATION AND CONCLUSIONS

8.1 DIAG-NLP3 evaluation

The evaluation of DIAG-NLP3 is still a work in progress. It requires a large data collection through experiments with different groups of students. This section presents some preliminary results of recent experiments carried out by other researchers at the NLP Lab at the University of Illinois at Chicago.

There are two main dimensions that are interesting to measure while evaluating an ITS; they are the *learning gain* and the *user preferences*. The *learning gain* can be calculated as the difference between two scores obtained by a student in a specific test, before and after using the ITS. So, to compare two different ITSs using learning gain, it is necessary to use two independent groups of students with homogeneous characteristics and make each group interact only with one of the systems. *User preferences* are collected making some students use all the different systems to compare, then giving them a questionnaire to fill.

Up to now, the data available is only about user preferences. Two distinct sessions of experiments have been run. One of them compares DIAG-NLP2 and DIAG-NLP3 against the original DIAG (without NLP extensions). The other one compares the three versions of DIAG-NLP, namely DIAG-NLP1, DIAG-NLP2, and DIAG-NLP3.

Let us examine the first set of results, shown in TABLE XIII and TABLE XIV. In these experiments, the comparison was between DIAG-Orig and DIAG-NLP2 and between DIAG-Orig and DIAG-NLP3. Both the NLP-enhanced systems score much higher than the original system, and DIAG-NLP3 is the one that scores higher. Notice that the preference in favor of DIAG-NLP3 is achieved mainly because of its naturalness and conciseness, as expected. It is not surprising that DIAG-NLP3 is not judged "more contentful" (i.e., providing more information) by users, because the quantity of information presented is in fact almost the same as in the original DIAG.

TABLE XIII. USER'S PREFERENCES BETWEEN THE ORIGINAL DIAG AND DIAG-NLP2.

	DIAG-Orig	***DIAG-NLP2***	***Neutral***
Number of preferences (total number of subjects: 25)	9	15	1
Total score obtained summing these grades: prefer a lot = 5 points prefer a little = 4 points	42	69	
Preference strength (Total score / # of preferences)	4.7	4.6	
Preference reason (subjects could circle more than one reason)			
More natural	2	6	
More concise	3	7	
Clearer	7	15	
More contentful	2	7	

TABLE XIV. USER'S PREFERENCES BETWEEN THE ORIGINAL DIAG AND DIAG-NLP3.

	DIAG-Orig	***DIAG-NLP3***	***Neutral***
Number of preferences (total number of subjects: 25)	7	17	1
Total score obtained summing these grades: prefer a lot = 5 points prefer a little = 4 points	32	80	
Preference strength (Total score / # of preferences)	4.6	4.7	
Preference reason (subjects could circle more than one reason)			
More natural	1	7	
More concise	1	11	
Clearer	4	9	
More contentful	4	4	

The other set of results is shown in TABLE XV and TABLE XVI. In these experiments, DIAG-NLP1 was preferred more than DIAG-NLP2, and DIAG-NLP2 was preferred more than DIAG-NLP3. We can compute the *preference strength* as the ratio of the preference score over the number of preferences expressed; when DIAG-NLP1 and DIAG-NLP2 are compared, they present the same preference strength (4.6); in the comparison between DIAG-NLP2 and DIAG-NLP3, DIAG-NLP3 shows a higher preference strength (4.7 against 4.4). Notice that we could not compare preference strength values across different tables. Although DIAG-NLP3's short answers might appear less preferred than DIAG-NLP1 and DIAG-NLP2 longer ones, a recent study argues that students are unlikely to read feedback answers longer than 3 lines (Heift 2001). So, there is the possibility that even though students say that they prefer DIAG-NLP1 and DIAG-NLP2 answers on a direct comparison, they would not read those answers in practice because they are too long. This interesting topic would require a longer study to be investigated.

TABLE XV. USER'S PREFERENCES BETWEEN DIAG-NLP2 AND DIAG-NLP3.

	DIAG-NLP2	***DIAG-NLP3***	***Neutral***
Number of preferences (total number of subjects: 23)	13	9	1
Total score obtained summing these grades: prefer a lot = 5 points prefer a little = 4 points	57	42	
Preference strength (Total score / # of preferences)	4.4	4.7	
Preference reason (subjects could circle more than one reason)			
More concise	4	5	
Clearer	3	6	
More useful	8	3	
More contentful	8	2	

TABLE XVI. USER'S PREFERENCES BETWEEN DIAG-NLP1 AND DIAG-NLP2.

	DIAG-NLP1	***DIAG-NLP2***	***Neutral***
Number of preferences (total number of subjects: 23)	14	9	0
Total score obtained summing these grades: prefer a lot = 5 points prefer a little = 4 points	64	41	
Preference strength (Total score / # of preferences)	4.6	4.6	
Preference reason (subjects could circle more than one reason)			
More concise	3	4	
Clearer	3	3	
More useful	7	9	
More contentful	2	3	

From this preliminary evaluation, we can see that users prefer a system with more natural and fluent language feedback. It is still not clear which version of the system is better; this judgement would require at least some experimental measures of students' learning gain, and we do not have this data at the present time. As already stated, the results showed here cannot be considered a full formal evaluation of the system; such an analysis is beyond the scope of this work.

8.2 **Conclusions and future work**

This thesis showed how an existing Intelligent Tutoring System's output can be improved with Natural Language Processing techniques, and how this improved system was designed using a five-step scientific approach (data collection, data annotation, data analysis, implementation, and evaluation).

The DIAG-NLP project was carried out by different researchers in multiple years; its conclusion will occur probably after the completion of DIAG-NLP3 (in its missing "Consult RU" part) and its formal evaluation. It will be important to extend the Sentence Planner to cover also *Consult RU* queries, because our statistics show that students prefer to ask DIAG about replaceable units rather than indicators (see Section 4.3). Since the original DIAG tells almost nothing about that kind of answers, while human tutors say a lot, it would be necessary to augment DIAG's knowledge. A possible meaningful source of information could be obtained taking into account the history of the interactions between the user and the system. Notice that the line between the NLP module and the ITS reasoning engine becomes blurred. To implement sophisticated discourse features it is not enough to enhance DIAG's "way of speaking," but it would be necessary to improve DIAG's "way of thinking" too. At this time, the improvement of *Consult RU* answers is under investigation by other researchers at the NLP Lab at the University of Illinois at Chicago.

A possible improvement to the current implementation of DIAG-NLP3 is to decouple the lexicalization process from the Sentence Planner. This would make it possible to have more linguistic

variety by using synonyms. At the present time, some of the lexemes to be used are decided by the Sentence Planner. For example, in the generated sentence “In the oil burner, check the units along the path of the oil and the burner motor,” the verb “check” is hard coded in the Sentence Planner, but logically it could be chosen by another module; in our example, the verb “check” could be replaced by “verify,” “see,” “look at,” or many other equivalent verbs.

A major issue for future research would be the development of methodologies and tools to speed up the data analysis process. Analyzing annotated data is actually a long, tedious, and error-prone process; an effective usage of semi-automatic techniques and machine learning could be a breakthrough in NLG software development process.

CITED LITERATURE

Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. 1995. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4(2):167-207

Bateman, J. 1996. KPML Development Environment. *Technical Report, IPSI, GMD, Darmstadt, Germany*.

Beck, J., Stern, M., and Haugsjaa, E. 1996. Applications of AI in Education. *ACM Crossroads, http://www.acm.org/crossaroads/xrds3-1/aied.html*

Brachman, R. J., McGuinness, D. L., Patel-Schneider, P. F., Alperin Resnick, L., and Borgida, A. 1990. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. P*rinciples of Semantic Networks, J. Sowa, ed., Morgan Kaufmann Publishers, Inc.,* 1991.

Broberg, A. 2000. Tools for learners as Knowledge Workers. *PhD thesis, Department of Computing Sciences, Umea University, Sweden*, 2000.

Dalianis, H. 1996. Concise Natural Language Generation from Formal Specifications. *PhD thesis*, April 1996.

Di Eugenio, B. 2001. Natural Language Processing for Computer-supported Instruction. *ACM Intelligence*, 12(4):22-32, Winter 2001.

Di Eugenio, B. and Trolio, M. J. 2000. Can simple Natural Language Generation improve Intelligent Tutoring Systems? *Building Dialogue Systems for Tutorial Applications (AAAI Fall Symposium)*, November 2000.

Di Eugenio, B., Glass, M., and Trolio, M. J. 2002. The DIAG experiments: Natural Language Generation for Intelligent Tutoring Systems. *The Second International Natural Language Generation Conference Proceedings*, July 2002.

Di Eugenio, B., Haller, S., Glass, M. 2003. Development and Evaluation of NL interfaces in a Small Shop. *2003 AAAI Spring Symposium on Natural Language Generation in Spoken and Written Dialogue*, Stanford, CA, March 2003.

Elhadad, M. 1993. FUF: the universal unifier - user manual version 5.2. *Columbia University, CUCS-038-91*.

Elhadad, M. and Robins, J. 1996. An overview of SURGE: a reusable comprehensive syntactic realisation component. *In Proceedings of the 8th International Workshop on Natural Language Generation (Demos and Posters)*, pages 1-4.

Evens, M. W., Spitkovsky, J., Boyle, P., Michael, J. A., Rovick, A. A. 1993. Synthesizing Tutorial Dialogues. *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pp. 137-140.

Evens, M. W., Brandle, S., Chang, R., Freedman, R., Glass, M., Lee, Y. H., Shim, L. S., Woo, C. W., Zhang, Y., Zhou, Y., Michael, J. A., Rovick, A. A. 2001. CIRCSIM-Tutor: An Intelligent Tutoring System Using Natural Language Dialogue. *12th Midwest AI and Cognitive Science Conference*, Oxford OH, 2001, pp. 16-23.

Glass, M. 1999. Broadening Input Understanding in a Language-Based Intelligent Tutoring System. *PhD Thesis. Illinois Institute of Technology*, May 1999.

Glass, M. and Di Eugenio, B. 2002. MUP: the UIC standoff markup tool. *In Third SIGDIAL Workshop on Discourse and Dialogue, Philadelphia. Association for Computational Linguistics*, 2002.

Glass, M., Raval, H., Di Eugenio, B., and Traat, M. 2002. The DIAG-NLP dialogues: coding manual. *Tech report, University of Illinois at Chicago, ref. UIC-CS 02-03*, 2002.

Graesser, A. C., VanLehn, K., Rosé, C. P., Jordan, P. W., and Harter, D. 2001. Intelligent tutoring systems with conversational dialogue. *AI Magazine*, 22, 39-51.

Haller, S. and Di Eugenio, B. 2002. Text Structuring to Improve the Presentation of Aggregated Content. *Tech report, University of Illinois at Chicago, ref. UIC-CS 02-02*, June 20, 2002.

Halliday, M. 1985. An Introduction to Functional Grammar. *Edward Arnold, London*.

Heift, T. 2001. Error-Specific and Individualized Feedback in a Web-based Language Tutoring System: Do They Read It? *ReCALL Journal, Volume 13 (2): 129-142, Cambridge University Press.*

Jurafsky, D. and Martin, J. H. 2000. Speech and Language Processing. *Prentice Hall.*

Kay, M. 1979. Functional Grammar. *In BLS-79*, Berkeley, CA, pp. 142-158.

Lavoie, B. and Rambow, O. 1997. A Fast and Portable Realizer for Text Generation Systems. *Proceedings of the Fifth Conference on Applied Natural Language Processing*, 1997.

Mel'cuk, I. and Pertsov, N. 1987. Surface Syntax of English: A Formal Model within the Meaning-Text Framework. *John Benjamins Publishing Company.*

Mel'cuk, I. 1988. Dependency Syntax: Theory and Practice. *State University of New York Press*.

Munro, A. 1994. Authoring interactive graphical models. In De Jong, T.; Towne, D. M.; and Spada, H., eds., *The Use of Computer Models for Explication, Analysis and Experiential Learning Sciences* 5(1):49-94.

Murray, T. and Woolf, B. 1992. Results of Encoding Knowledge with Tutor Construction Tools. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 17-23.

Reape, M. and Mellish, C. 1998. Just what is aggregation anyway? *Proceedings of the European Workshop on Natural Language Generation*, Toulouse, France, 1998.

Reiter, E. and Dale, R. 1997. Building Applied Natural Language Generation Systems. *Natural Language Engineering Journal*, vol. 3, 1997.

Shapiro, S. and Rapaport, W. 1992. The SnePS Family. *Computers and Mathematics with Applications, Special Issue on Semantic Networks in Artificial Intelligence*, Part1, 23(2-5).

Shute, V. J., and Psotka, J. 1996. Intelligent tutoring systems: Past, Present and Future. In *D. Jonassen (Ed.), Handbook of Research on Educational Communications and Technology* : Scholastic Publications.

Towne, D. M. 1997a. Approximate reasoning techniques for intelligent diagnostic instruction. *International Journal of Artificial Intelligence in Education*, vol. 8, pp. 262-283.

Towne, D. M. 1997b. Intelligent diagnostic tutoring using qualitative symptom information. In *AAAI97 Fall Symposium on ITS Authoring Tools*.

APPENDIX A

****
Class Aggregand in Package diagnlp3.agg
****

```
package diagnlp3.agg;

import java.util.*;

/**
 * This class represents a generic aggregand object. An aggregand
 * object is composed by a name and a set of relations.
 * In the actual implementation, all the relations are taken from
 * a static knowledge base, passed as parameter to its contructor.
 * @author Davide Fossati
 */

public class Aggregand {
    /**
     * This constructor builds an Aggregand with a given name,
     * and infer all of its relations from a given StaticKnowledgeBase.
     * @param name the name of this Aggregand
     * @param kb the StaticKnowledgeBase to use for finding the relations
     */
    public Aggregand(String name, StaticKnowledgeBase kb) {
        this.name = name;
        this.relations = kb.getRelations(name);
    }


    /**
     * This constructor builds an Aggregand with a given name
     * and a given Vector of relations.
     * @param name the name of this Aggregand
     * @param relations a Vector of Relation objects
     */
    public Aggregand(String name, Vector relations) {
        this.name = name;
        this.relations = relations;
    }


    public String toString() {
        return name+":"+relations;
    }


    /**
     * Returns the name of this Aggregand.
     * @return a String representing the name of this Aggregand
     */
    public String getName() {
        return name;
    }

    /**
     * Sets a new name to this Aggregand.
     * @param newName the new name of this Aggregand
     */
    public void setName(String newName) {
        this.name = newName;
    }

    /**
     * Returns all the relations of this Aggregand.
```

APPENDIX A (continued)

```
     * @return a Vector containing the relations (of class Relation) of this Aggregand
     */
    public Vector getRelations() {
        return relations;
    }


    /**
     * Adds a relation to this Aggregand.
     * @param r the Relation to be added
     */
    public void addRelation(Relation r) {
        relations.add(r);
    }

    /**
     * Removes a relation from this Aggregand.
     * @param relName the name of the relation to be removed
     */
    public void removeRelation(String relName) {
        int n = relations.size();
        for(int i = 0; i < n; i++) {
            Relation r = (Relation)relations.elementAt(i);
            if(r.getName().equals(relName)) {
                relations.remove(i);
                i = n;
            }
        }
    }

    /**
     * Finds a specific relation in the relation set of this Aggregand.
     * @param relName the name of the relation to be searched
     * @return the requested relation, or null if that relation is not found
     */
    public Relation getRelation(String relName) {
        int n = relations.size();
        for(int i = 0; i < n; i++) {
            Relation r = (Relation)relations.elementAt(i);
            if(r.getName().equals(relName)) {
                return r;
            }
        }
        return null;
    }

    private String name;

    private Vector relations; //elements: Relation objects
}
```

APPENDIX A (continued)

****
Class Aggregation in Package diagnlp3.agg
****

```
package diagnlp3.agg;

import java.util.*;

/**
 * This class represents an aggregation.
 * An Aggregation object in composed by a Vector of Aggregand objects
 * and a Vector of aggregator relations (Relation objects).
 * @author Davide Fossati
 */

public class Aggregation {
    /**
     * Builds an Aggregation given its aggregands and its aggregators.
     * @param aggregands a Vector of Aggregand objects
     * @param aggregators a Vector of Relation objects
     */
    public Aggregation(Vector aggregands, Vector aggregators) {
        this.aggregands = aggregands;
        this.aggregators = aggregators;
    }


    /**
     * Creates an Aggregation object from a set of aggregands.
     * @param aggregands a Vector containing some Aggregand objects
     */
    public Aggregation(Vector aggregands) {
        this.aggregators = new Vector();
        this.aggregands = aggregands;

        if(aggregands.size() > 0) {
            boolean flag;
            Vector relations1 = ((Aggregand)aggregands.firstElement()).getRelations();

            Iterator relIter = relations1.iterator();
            while(relIter.hasNext()) {
                Relation rel = (Relation)relIter.next();
                flag = true;

                Iterator aggIter = aggregands.iterator();
                while(aggIter.hasNext() && flag == true) {
                    Aggregand agg = (Aggregand)aggIter.next();
                    if(!agg.getRelations().contains(rel)) {
                        flag = false;
                    }
                }
                if(flag) {
                    aggregators.add(rel);
                }
            }
        }
    }


    /**
     * Returns the aggregands of this Aggregation.
     * @return a Vector of Aggregand objects
     */
    public Vector getAggregands() {
```

APPENDIX A (continued)

```
    return aggregands;
}


/**
 * Returns the aggregator relations of this Aggregation.
 * @return a Vector of Relation objects
 */
public Vector getAggregators() {
    return aggregators;
}

public String toString() {
    return "Aggregands: " + aggregands +
        ", Aggregators: " + aggregators;
}


/**
 * Checks if a given relation is part of the aggregators of this Aggregation,
 * and returns it.
 * @param relationName the name of a Relation
 * @return the actual Relation if there is a match, null otherwise
 **/
public Relation getAggregator(String relationName) {
    Iterator it = aggregators.iterator();
    while(it.hasNext()) {
        Relation r = (Relation)it.next();
        if(r.getName().equals(relationName)) {
            return r;
        }
    }
    return null;
}

/**
 * Displays this object in a human-readable format
 * on the standard output.
 */
public void display() {
    Iterator it;
    System.out.println("Aggregands:");
    it = aggregands.iterator();
    while(it.hasNext()) {
        Aggregand agg = (Aggregand)it.next();
        System.out.println("  " + agg.getName());
    }

    System.out.println("Aggregators:");
    it = aggregators.iterator();
    while(it.hasNext()) {
        Relation agg = (Relation)it.next();
        System.out.println("  " + agg);
    }
}


/**
 * Checks if this Aggregation dominates another Aggregation.
 * @param agg the Aggregation to be checked for domination
 * @return true if this Agrregation dominates the other one, false otherwise
 */
public boolean dominate(Aggregation agg) {
    if(aggregands.containsAll(agg.getAggregands())
```

APPENDIX A (continued)

```
            && aggregators.containsAll(agg.getAggregators())
            ) {
             return true;
        }
        else {
            return false;
        }
    }

    private Vector aggregands; //elements: Aggregand objects
    private Vector aggregators; //elements: Relation objects

}
```

APPENDIX A (continued)

****
Class Aggregator in Package diagnlp3.agg
****

```
package diagnlp3.agg;

import diagnlp3.sel.*;
import diagnlp3.misc.*;

import java.sql.*;
import java.util.*;

/**
 * This class provides several aggregation functions.
 * @author Davide Fossati
 */

public class Aggregator {
    /**
     * main method, used for testing and debug.
     */
    public static void main(String argv[]) {
        SentenceBuilder sb = new SentenceBuilder();
        Aggregation myAgg;

        if(argv.length > 0) {
            StaticKnowledgeBase myKB = new StaticKnowledgeBase();

            Vector allAggregands = new Vector(); //elements: Aggregand objects

            for(int i = 0; i < argv.length; i++) {
                allAggregands.add(new Aggregand(argv[i], myKB));
            }

            Vector finalAggregations = aggregate(allAggregands);

            Iterator fit = finalAggregations.iterator();
            while(fit.hasNext()) {
                Aggregation agg = (Aggregation)fit.next();
                agg.display();
                System.out.println();

                SentenceStructure ss = sb.renderAggregation(agg);
                String mySentence = sb.renderSentenceStructure(ss);
                System.out.println(mySentence);
                System.out.println();
            }
            myAgg = (Aggregation)finalAggregations.firstElement();
        }
        else {
            myAgg = BuildStubs.buildAggregation();
            myAgg.display();
            System.out.println();

            SentenceStructure ss = sb.renderAggregation(myAgg);
            String mySentence = sb.renderSentenceStructure(ss);
            System.out.println(mySentence);
        }
    }


    /**
     * Builds a Vector of Aggregation objects. It builds the power set
     * of the aggregands, and eliminates the dominated subsets.
```

APPENDIX A (continued)

```
 * @param allAggregands a Vector of Aggregand objects
 * @return a Vector of Aggregation objects
 */
public static Vector aggregate(Vector allAggregands) {
    if(allAggregands.size() > 0) {
        Vector aggPowerSet = Utility.buildPowerSet(allAggregands);
        Vector finalAggregations = new Vector();
        Iterator it = aggPowerSet.iterator();
        while(it.hasNext()) {
            Vector aggregands = (Vector)it.next();

            Aggregation myAggregation = new Aggregation(aggregands);
            Iterator finalIt = finalAggregations.iterator();
            boolean dominated = false;
            while(finalIt.hasNext() && dominated == false) {
                Aggregation agg = (Aggregation)finalIt.next();
                if(agg.dominate(myAggregation)) {
                    dominated = true;
                }
                else if(myAggregation.dominate(agg)) {
                    finalIt.remove();
                }
            }
            if(dominated == false) {
                finalAggregations.add(myAggregation);
            }
        }

        return finalAggregations;
    }
    else {
        return new Vector();
    }
}


/**
 * Builds a partition of the given set of Aggregands by the values of
 * a given Relation name
 * @param aggregands a Vector of Aggregand objects
 * @param relationName the name of the discriminating Relation
 * @return a Vector of Vectors of Aggregands, containing the desired partitions;
 *         the first subvector contains always the aggregands that do not have
 *         a Relation with the given name
 */
public static Vector partitionBy(Vector aggregands, String relationName) {
    Vector result = new Vector();
    Vector uniqueRelations = new Vector();

    Iterator it = aggregands.iterator();
    Vector noRelation = new Vector();
    while(it.hasNext()) {
        Aggregand agg = (Aggregand)it.next();
        if(agg.getRelation(relationName) == null) {
            noRelation.add(agg);
        }
    }

    result.add(noRelation);

    Iterator it2 = aggregands.iterator();
    Vector relationInstances = new Vector();
    while(it2.hasNext()) {
        Aggregand agg = (Aggregand)it2.next();
```

APPENDIX A (continued)

```
            Relation rel = agg.getRelation(relationName);
            if(rel != null) {
                if(!relationInstances.contains(rel)) {
                    relationInstances.add(rel);
                }
            }
        }

        Iterator it3 = relationInstances.iterator();
        while(it3.hasNext()) {
            Relation rel = (Relation)it3.next();
            Vector aggSet = new Vector();
            Iterator it4 = aggregands.iterator();
            while(it4.hasNext()) {
                Aggregand agg = (Aggregand)it4.next();
                Relation rel2 = agg.getRelation(relationName);
                if(rel2 != null) {
                    if(rel2.equals(rel)) {
                        aggSet.add(agg);
                    }
                }
            }
            result.add(aggSet);
        }

        return result;
    }


}
```

APPENDIX A (continued)

****
Class Relation in Package diagnlp3.agg
****

```
package diagnlp3.agg;

import java.util.*;

/**
 * This class represents a relation.
 * A relation is a named tuple with an arbitrary number of named elements.
 * @author Davide Fossati
 */

public class Relation {
    /**
     * Builds a relation object, given all its attributes.
     * @param relationName the name of this relation
     * @param type the type of this relation. It can be either "static" or "dynamic".
     *             Note: this attribute will be probably removed in future
     * @param paramNames a Vector of Strings representing the name of the parameters
of
     *                   this relation
     * @param paramValues a Vector of Strings representing the values of the
parameters.
     */
    public Relation(String relationName, String type, Vector paramNames, Vector
paramValues) {
        this.relationName = relationName;
        this.type = type;
        this.paramNames = paramNames;
        this.paramValues = paramValues;
    }


    /**
     * Returns the name of this Relation.
     * @return a String representing  the name of this Relation
     */
    public String getName() {
        return relationName;
    }

    /**
     * Sets a new name to this Relation.
     * @param newName the new name of this Relation
     */
    public void setName(String newName) {
        this.relationName = newName;
    }

    /**
     * Returns the type of this Relation.
     * @return a String representing  the type of this Relation ("static" or
"dynamic")
     */
    public String getType() {
        return type;
    }


    /**
     * Returns the number of parameters in this Relation.
     * @return the number of parameters in this Relation
```

APPENDIX A (continued)

```
     */
    public int getParamNumber() {
        return paramNames.size();
    }


    /**
     * Returns the names of the parameters in this Relation.
     * @return a Vector of Strings representing the name of the parameters in this
Relation
     */
    public Vector getParamNames() {
        return paramNames;
    }


    /**
     * Returns the values of all the parameters in this Relation.
     * @return a Vector of Strings representing the value of all the parameters in
this Relation
     */
    public Vector getParamValues() {
        return paramValues;
    }

    public String getValue(int index) {
        if(index == -1) {
            return "";
        }
        else {
            return (String)paramValues.elementAt(index);
        }
    }


    /**
     * Returns the value of a given parameter in this Relation.
     * @return a String representing the value of the desired parameter
     */
    public String getValue(String param) {
        int index = paramNames.indexOf(param);
        if(index == -1) {
            return "";
        }
        else {
            return (String)paramValues.elementAt(index);
        }
    }


    public boolean equals(Object o) {
        if(!this.getClass().equals(o.getClass())) {
            return false;
        }
        Relation r = (Relation)o;
        if(relationName.equals(r.getName()) &&
           type.equals(r.getType()) &&
           paramNames.equals(r.getParamNames()) &&
           paramValues.equals(r.getParamValues())) {
            return true;
        }
        else {
            return false;
        }
```

APPENDIX A (continued)

```
    }


    public String toString() {
        return "Relation(" + relationName + ":" + type + ":" +
            paramNames + ":" + paramValues + ")";
    }


    private String relationName;
    private String type;
    private Vector paramNames; // elements: Strings
    private Vector paramValues; // elements: Strings

}
```

APPENDIX A (continued)

```
      ***************************************************************************
                Class StaticKnowledgeBase in Package diagnlp3.agg
      ***************************************************************************

package diagnlp3.agg;

import diagnlp3.misc.Constants;
import diagnlp3.misc.Utility;

import java.util.*;
import java.sql.*;

/**
 * This class provides an interface to an external knowledge base.
 * A knowledge base contains a set of relations. Those relations
 * can be retrieved with the methods provided by this class.
 * @author Davide Fossati
 */

public class StaticKnowledgeBase {
    /**
     * Creates this StaticKnowledgeBase object, establishing the connection
     * with the underlying knowledge base.
     */
    public StaticKnowledgeBase() {
        dbStatement1 = null;
        dbStatement2 = null;

        try {
            // loads the driver
            Class.forName(Constants.jdbcDriver);

            // makes the connection
            Connection con = DriverManager.getConnection(Constants.kbDatabase, "",
"");

            // creates a statement
            dbStatement1 = con.createStatement();
            dbStatement2 = con.createStatement();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }


    /**
     * Queries the knowledge base for all the static relations associated to a given
object.
     * Note: the implementation of this method returns a set of relations that are
distinct
     * by name (i.e. relation parameters should not be multi-valued).
     * @param objectName the name of the object to be queried
     * @return a Vector of Relation objects
     */
    public Vector getRelations(String objectName) {
        Vector result = new Vector(); // elements: Relation objects

        try {
            String query1 =
                "SELECT name, param_number, param_names " +
                "FROM objects_relations, relations " +
                "WHERE name = relation AND object = '" + objectName + "'" +
                "      AND type = 'static'";
```

APPENDIX A (continued)

```
            // executes the query
            ResultSet rs1 = dbStatement1.executeQuery(query1);

            while(rs1.next()) {
                String relName = rs1.getString("name");
                int parNum = rs1.getInt("param_number");
                String parNames = rs1.getString("param_names");

                String query2 =
                    "SELECT " + parNames +
                    " FROM " + relName +
                    " WHERE object = '" + objectName + "'";

                ResultSet rs2 = dbStatement2.executeQuery(query2);

                Vector values = new Vector();
                String[] v = new String[parNum];

                while(rs2.next()) {  //it's a while loop, but retains only 1 element
                    for(int i = 0; i < parNum; i++) {
                        v[i] = rs2.getString(i+1);
                    }
                }
                for(int i = 0; i < parNum; i++) {
                    values.add(v[i]);
                }

                result.add(new Relation(relName, "static",
Utility.splitString(parNames), values));
            }

        } catch (Exception ex) {
            ex.printStackTrace();
        }

        return result;
    }


    /**
     * Queries the knowledge base for a single relation, given a pair object-relation
name.
     * If no such a relation exists, returns null.
     * @param objectName the name of the object to be queried
     * @param relationName the name of the desired relation
     * @return a Relation object, or null if the requested relation is not found
     */
    public Relation getSingleRelation(String objectName, String relationName) {
        Relation result = null;

        try {
            String query1 =
                "SELECT param_number, param_names " +
                "FROM relations " +
                "WHERE name = '" + relationName + "'";

            // executes the query
            ResultSet rs1 = dbStatement1.executeQuery(query1);

            if(rs1.next()) {
                int parNum = rs1.getInt("param_number");
                String parNames = rs1.getString("param_names");

                String query2 =
```

APPENDIX A (continued)

```
                    "SELECT " + parNames +
                    " FROM " + relationName +
                    " WHERE object = '" + objectName + "'";

                ResultSet rs2 = dbStatement2.executeQuery(query2);

                Vector values = new Vector();
                String[] v = new String[parNum];

                while(rs2.next()) {  //it's a while loop, but retains only 1 element
                    for(int i = 0; i < parNum; i++) {
                        v[i] = rs2.getString(i+1);
                    }
                }
                for(int i = 0; i < parNum; i++) {
                    values.add(v[i]);
                }

                result = new Relation(relationName, "unspecified",
Utility.splitString(parNames), values);
            }

        } catch (Exception ex) {
            ex.printStackTrace();
        }

        return result;
    }


    private Statement dbStatement1;
    private Statement dbStatement2;

}
```

APPENDIX A (continued)

```
      ************************************************************************
               Class ConsultIndicatorDiag in Package diagnlp3.consult
      ************************************************************************

package diagnlp3.consult;

import java.util.*;

/**
 * This class stores the information provided by Vivids-DIAG
 * in a single "consult indicator" answer
 * @author Davide Fossati
 */

public class ConsultIndicatorDiag {
    public ConsultIndicatorDiag() {
        ruName = new Vector();
        ruFufer1 = new Vector();
        ruFufer2 = new Vector();
    }

    /**
     * Returns the name of the indicator being consulted.
     * @return the name of the indicator being consulted
     */
    public String getIndicatorName() {
        return indicatorName;
    }


    /**
     * Returns the actual state of the indicator being consulted.
     * @return the actual state of the indicator being consulted
     */
    public String getIndicatorState() {
        return indicatorState;
    }


    /**
     * Returns the normal state of the indicator being consulted.
     * @return the normal state of the indicator being consulted
     */
    public String getIndicatorNormalState() {
        return indicatorNormalState;
    }


    /**
     * Returns the name of a replaceable unit in the list.
     * @param index the index of the desired replaceable unit
     * @return the name of the requested replaceable unit
     */
    public String getReplUnitName(int index) {
        return (String)ruName.elementAt(index);
    }


    /**
     * Returns the first "fufer" word ("always", "never", etc)
     * of a replaceable unit in the list.
     * @param index the index of the desired replaceable unit
     * @return the first "fufer" word of the requested replaceable unit
     */
```

APPENDIX A (continued)

```
    public String getReplUnitFufer1(int index) {
        return (String)ruFufer1.elementAt(index);
    }


    /**
     * Returns the second "fufer" word ("normal", "abnormal", etc)
     * of a replaceable unit in the list.
     * @param index the index of the desired replaceable unit
     * @return the second "fufer" word of the requested replaceable unit
     */
    public String getReplUnitFufer2(int index) {
        return (String)ruFufer2.elementAt(index);
    }


    /**
     * Sets the name of the indicator in this consult.
     * @param newName the new name of the indicator
     */
    public void setIndicatorName(String newName) {
        indicatorName = newName;
    }


    /**
     * Sets the actual state of the indicator in this consult.
     * @param newState the new state of the indicator
     */
    public void setIndicatorState(String newState) {
        indicatorState = newState;
    }


    /**
     * Sets the normal state of the indicator in this consult.
     * @param newNormalState the new normal state of the indicator
     */
    public void setIndicatorNormalState(String newNormalState) {
        indicatorNormalState = newNormalState;
    }


    /**
     * Adds a replaceable unit to the list of this consult.
     * @param name the name of the new replaceable unit
     * @param fufer1 the first "fufer" word ("always", "never", etc)  of the new
replaceable unit
     * @param fufer1 the second "fufer" word ("abnormal", "normal", etc)  of the new
replaceable unit
     */
    public void addReplUnit(String name, String fufer1, String fufer2) {
        ruName.add(name);
        ruFufer1.add(fufer1);
        ruFufer2.add(fufer2);
    }


    /**
     * Displays this object in a human readable format.
     */
    public void display() {
        System.out.println("Mode name: " + modeName);
        System.out.println("Indicator name: " + indicatorName);
```

APPENDIX A (continued)

```
        System.out.println("Indicator state: " + indicatorState);
        System.out.println("Indicator normal state: " + indicatorNormalState);
        System.out.println("Indicator normality: " + indicatorNormality());

        int n = ruName.size();

        for(int i = 0; i < n; i++) {
            System.out.println("ReplUnit name: " + ruName.elementAt(i));
            System.out.println("ReplUnit fufer1: " + ruFufer1.elementAt(i));
            System.out.println("ReplUnit fufer2: " + ruFufer2.elementAt(i));
        }
    }


    /**
     * Checks if this ConsultIndicator is consistent. In order to be consistent,
     * a ConsultIndicator must have the same number of names, fufer1, and fufer2
elements.
     * @return true if this ConsultIndicator is consistent, false otherwise
     */
    public boolean isConsistent() {
        int n = ruName.size();

        if(n != ruFufer1.size() || n != ruFufer2.size()) {
            return false;
        }
        else {
            return true;
        }
    }

    /**
     * Returns the "normality" of the indicator of this consult.
     * Three values are possible:
     * "normality", if the actual state is the same as the normal state;
     * "abnormality1", if the actual state is greater than the normal state;
     * "abnormality2", if the actual state is less than the normal state.
     * If the states are not expressed by numbers, there are only two values
     * ("normality" and "abnormality1").
     * @return "normality", "abnormality1", or "abnormality2"
     */
    public String indicatorNormality() {
        if(indicatorState.equals(indicatorNormalState)) {
            return "normality";
        }
        try {
            double state = Double.valueOf(indicatorState).doubleValue();
            double normalState = Double.valueOf(indicatorNormalState).doubleValue();
            if(state > normalState) {
                return "abnormality1";
            }
            else {
                return "abnormality2";
            }
        }
        catch (NumberFormatException ex) {
            return "abnormality1";
        }
    }

    public String modeName = null;

    public String indicatorName = null;
    public String indicatorState = null;
```

APPENDIX A (continued)

```
    public boolean indicatorStateNumber = false;
    public String indicatorNormalState = null;
    public boolean indicatorNormalStateNumber = false;

    public Vector ruName;   // elements: Strings
    public Vector ruFufer1; // elements: Strings
    public Vector ruFufer2; // elements: Strings
}
```

APPENDIX A (continued)

```
      ************************************************************************
              Class ConsultIndicatorNLP in Package diagnlp3.consult
      ************************************************************************

package diagnlp3.consult;

import diagnlp3.misc.Constants;
import diagnlp3.agg.*;
import diagnlp3.sel.*;
import diagnlp3.real.*;

import java.util.*;
import java.io.*;

/**
 * This class generates the answer for a request of type
 * "Consult Indicator". It uses only the information provided
 * by Diag (i.e. the text file generated at run-time by Diag).
 * @author Davide Fossati
 */

public class ConsultIndicatorNLP {
    /**
     * Main method, used for testing and debug
     */
    public static void main(String argv[]) {
        SurfaceRealizer realizer = new SurfaceRealizer();
        ConsultIndicatorNLP myConsult = new ConsultIndicatorNLP(realizer);
        myConsult.generateResponse();
    }


    /**
     * Initializes this object. The text file provided by Diag
     * is parsed at this time.
     */
    public ConsultIndicatorNLP(SurfaceRealizer realizer) {
        this.realizer = realizer;
        knowledgeBase = new StaticKnowledgeBase();

        responseSS = new Vector();
        responseString = new Vector();

        DiagCommunicator myDC = new DiagCommunicator();
        consultIndicatorDIAG = myDC.parseConsultIndicatorFile();
    }


    /**
     * Performs all the processing required to generate the actual answer
     */
    public void generateResponse() {
        SentenceBuilder sb = new SentenceBuilder();

        // Talk about the indicator

        SentenceStructure indicatorSS = buildIndicatorSS();
        responseSS.add(indicatorSS);
        responseString.add(sb.renderSentenceStructure(indicatorSS, realizer));


        // If the indicator is abnormal, talk about the replaceable units

        if(!consultIndicatorDIAG.indicatorNormality().equals("normality")) {
```

APPENDIX A (continued)

```
            Vector replUnits = new Vector();

            int n = consultIndicatorDIAG.ruName.size();
            for(int i = 0; i < n; i++) {
                if((((String)consultIndicatorDIAG.ruFufer1.elementAt(i)).equals("alway
s") &&
                    ((String)consultIndicatorDIAG.ruFufer2.elementAt(i)).equals("abnor
mal")) ||
                   (((String)consultIndicatorDIAG.ruFufer1.elementAt(i)).equals("never
") &&
                    ((String)consultIndicatorDIAG.ruFufer2.elementAt(i)).equals("norma
l"))) {

                    Aggregand ru = new
Aggregand((String)consultIndicatorDIAG.ruName.elementAt(i), knowledgeBase);
                    replUnits.add(ru);
                }
            }

            Vector partitionByPlace = Aggregator.partitionBy(replUnits, "place");
            int placePartitionCardinality = partitionByPlace.size();

            for(int i = 1; i < placePartitionCardinality; i++) {
                Vector replUnitsInTheSamePlace =
(Vector)partitionByPlace.elementAt(i);

                Aggregand anAggregand =
(Aggregand)replUnitsInTheSamePlace.firstElement();
                Relation relationPlace = anAggregand.getRelation("place");
                CommonNoun place = new CommonNoun(relationPlace.getValue("location") +
",");
                place.setPreposition("in");

                SentenceStructure replUnitSS = new SentenceStructure();
                replUnitSS.preVerbal = place;


                Vector partitionByFunctionalAggregation =
Aggregator.partitionBy(replUnitsInTheSamePlace,
                                                                                "func
tional_aggregation");
                Vector aggUnits = new Vector(); //elements: CommonNoun objects
                int functionalPartitionCardinality =
partitionByFunctionalAggregation.size();
                for(int j = 1; j < functionalPartitionCardinality; j++) {
                    Vector nouns =
resolveFunctionalAggregation((Vector)partitionByFunctionalAggregation.elementAt(j));
                    aggUnits.addAll(nouns);
                }

                aggUnits.addAll(resolveNames((Vector)partitionByFunctionalAggregation.
firstElement()));

                replUnitSS.verb = new Verb("check");
                ((Verb)replUnitSS.verb).mood = "imp";
                int numReplUnits = aggUnits.size();
                if(numReplUnits > 0) {
                    SyntacticElement object =  (CommonNoun)aggUnits.firstElement();
                    replUnitSS.object = object;
                    for(int j = 1; j < numReplUnits; j++) {
                        CommonNoun coord = (CommonNoun)aggUnits.elementAt(j);
                        object.addCoordinate(coord);
                    }
                }
```

APPENDIX A (continued)

```
                String responseReplUnit = sb.renderSentenceStructure(replUnitSS,
realizer);
                responseString.add(responseReplUnit);
            }
        }

        writeResponse();
        displayResponse();
    }


    /**
     * Prints the answer on the standard output
     */
    public void displayResponse() {
        Iterator it = responseString.iterator();
        while(it.hasNext()) {
            String s = (String)it.next();
            System.out.println(s);
        }
    }


    /**
     * Writes the answer on a file. That file will be eventually read by Diag and
     * displayed on the screen.
     */
    public void writeResponse() {
        try {
            FileWriter responseFileWriter = new FileWriter(Constants.genDialogFile);

            Iterator it = responseString.iterator();
            while(it.hasNext()) {
                String s = (String)it.next();
                responseFileWriter.write(s);
                responseFileWriter.write('\n');
            }
            responseFileWriter.close();
        } catch (Exception ex) { ex.printStackTrace(); }
    }


    /**
     * Builds the structure of the first sentence of the
     * answer. This sentence talks about an indicator.
     * @return a new SentenceStructure object
     */
    public SentenceStructure buildIndicatorSS() {
        Aggregand indicator = new Aggregand(consultIndicatorDIAG.indicatorName,
knowledgeBase);
        Aggregand indication = resolveImplicitness(indicator);
        Relation normality = knowledgeBase.getSingleRelation(indication.getName(),
"normality");

        SentenceStructure indicatorSS = new SentenceStructure();

        indicatorSS.subject = new CommonNoun(indication.getName());
        ((CommonNoun)indicatorSS.subject).article = "def";
        indicatorSS.verb = new Verb("be");
        indicatorSS.object = new
Adjective(normality.getValue(consultIndicatorDIAG.indicatorNormality()));

        return indicatorSS;
```

APPENDIX A (continued)

```
    }


    /**
     * Process an implicit aggregand and makes it explicit. For example,
     * an indicator like "water temperature gauge" will be dereferenced as
     * "water".
     * @param implicitAggregand the Aggregand to be dereferenced
     * @return a new Aggregand
     */
    public Aggregand resolveImplicitness(Aggregand implicitAggregand) {
        Relation implicitness = implicitAggregand.getRelation("implicitness");
        if(implicitness != null) {
            Aggregand explicitAggregand = new
Aggregand(implicitness.getValue("reference"),
                                                        knowledgeBase);
            return explicitAggregand;
        }
        else {
            return implicitAggregand;
        }
    }


    /**
     * Process a set of Aggregands according to the information given
     * by the Relation named "functional_aggregation". These Aggregands are
     * processed in this way: if the number of elements in the actual set is less
     * than 50% of the maximum number of elements of the full aggregating superset,
     * then nothing happens; if the actual set contains 100% of the elements of the
     * superset, they are replaced with the name of the superset; if the number of
     * elements in this set is between 50% and 100%, those elements are
     * replaced with the name of the superset, quantified by "some_of".
     * @param a vector of Aggregands
     * @return a vector of Aggregands
     */
    public Vector resolveFunctionalAggregation(Vector aggregands) {
        Vector answer = new Vector();

              if(!aggregands.isEmpty()) {
            Relation rel =
((Aggregand)aggregands.firstElement()).getRelation("functional_aggregation");
                int fullCardinality = Integer.parseInt(rel.getValue("cardinality"));
            int actualCardinality = aggregands.size();

            if(actualCardinality < fullCardinality / 2) {
                Iterator it=aggregands.iterator();
                while(it.hasNext()) {
                    Aggregand agg = (Aggregand)it.next();
                    CommonNoun element = new CommonNoun(agg.getName());
                    answer.add(element);
                }
            }
            else {
                CommonNoun element = new CommonNoun(rel.getValue("aggregator"));

                if(actualCardinality < fullCardinality) {
                    element.setPreposition("some_of");
                }

                answer.add(element);
            }
        }
        return answer;
```

APPENDIX A (continued)

```
    }


    /**
     * Builds a set of CommonNoun objects representing the names of some
     * Aggregand objects.
     * @param a vector of Aggregand objects
     * @return a vector of CommonNoun objects
     */
    public Vector resolveNames (Vector aggregands) {
        Vector answer = new Vector();

        Iterator it=aggregands.iterator();
        while(it.hasNext()) {
            Aggregand agg = (Aggregand)it.next();
            CommonNoun element = new CommonNoun(agg.getName());
            answer.add(element);
        }

        return answer;
    }


    private ConsultIndicatorDiag consultIndicatorDIAG;
    private StaticKnowledgeBase knowledgeBase;
    private SurfaceRealizer realizer;

    private Vector responseSS;     // elements: SentenceStructure objects
    private Vector responseString; // elements: String objects
}
```

APPENDIX A (continued)

```
**************************************************************************
            Class DiagCommunicator in Package diagnlp3.consult
**************************************************************************

package diagnlp3.consult;

import diagnlp3.misc.Constants;

import java.io.*;
import java.util.*;

/**
 * This class interacts with the Vivids-DIAG system.
 * @author Davide Fossati
 */

public class DiagCommunicator {
    /**
     * Main method, used for testing and debug
     */
    public static void main(String argv[]) {
        DiagCommunicator myDC = new DiagCommunicator();
        myDC.parseConsultIndicatorFile().display();
    }


    public DiagCommunicator() {
    }


    /**
     * Parses a Consult Indicator file built by DIAG, and creates
     * a ConsultIndicator structure containing the same information.
     * Note: this method is strongly dependent on the format of the file
     * provided by DIAG.
     * @return a ConsultIndicator object
     */
    public ConsultIndicatorDiag parseConsultIndicatorFile() {
        ConsultIndicatorDiag myConsult = new ConsultIndicatorDiag();

        try {
            FileReader fileReader = new FileReader(Constants.attrsFile);
            LineNumberReader lineReader = new LineNumberReader(fileReader);
            String line;
            boolean ruMode = false;
            boolean indicatorMode = true;
            String ruName = null;
            String ruFufer1;
            String ruFufer2;

            while((line = lineReader.readLine()) != null) {
                if(line.equals("ConsultIndicator ReplUnit")) {
                    indicatorMode = false;
                    ruMode = true;
                }
                else if(line.equals("ConsultIndicator Indicator")) {
                    indicatorMode = true;
                    ruMode = false;
                }
                else if(ruMode == true) {
                    if(line.startsWith("name")) {
                        ruName = replaceSpaces(line.substring(5).toLowerCase());
                    }
                    else if(line.startsWith("fufer")) {
```

APPENDIX A (continued)

```
                        StringTokenizer tokenizer = new
StringTokenizer(line.substring(6).toLowerCase());
                        ruFufer1 = tokenizer.nextToken();
                        ruFufer2 = tokenizer.nextToken();
                        myConsult.addReplUnit(ruName, ruFufer1, ruFufer2);
                    }
                }
                else if(indicatorMode == true) {
                    if(line.startsWith("name")) {
                        myConsult.indicatorName =
replaceSpaces(line.substring(5).toLowerCase());
                    }
                    else if(line.startsWith("state")) {
                        myConsult.indicatorState = line.substring(6).toLowerCase();
                    }
                    else if(line.startsWith("modeName")) {
                        myConsult.modeName = line.substring(9).toLowerCase();
                    }
                    else if(line.startsWith("normalState")) {
                        myConsult.indicatorNormalState =
line.substring(12).toLowerCase();
                    }
                }
            }
        }
        catch (Exception ex) { ex.printStackTrace(); }

        return myConsult;
    }


    /**
     * Replace all the spaces in a given string with the character '_'
     * @param s the input String
     * @return the modified String
     */
    public String replaceSpaces(String s) {
        StringBuffer sb = new StringBuffer(s);
        int n = sb.length();
        for(int i = 0; i < n; i++) {
            if(sb.charAt(i) == ' ') {
                sb.setCharAt(i, '_');
            }
        }
        return sb.toString();
    }
}
```

APPENDIX A (continued)

****
Class Constants in Package diagnlp3.misc
****

```
package diagnlp3.misc;

/**
 * This class defines all the constants (like filenames, etc)
 * used in the DiagNLP3 system.
 * @author Davide Fossati
 */

public class Constants {
    /** Name of the KB data source */
    public static String kbDatabase =
"jdbc:mysql://localhost/homeheat?user=hheat&password=diag";

    /** Name of the JDBC driver used to access the database */
    public static String jdbcDriver = "org.gjt.mm.mysql.Driver";

    /** Path of the file where DIAG stores its consult data */
    public static String attrsFile = "/space/hheat/hh/homehtStuff/attrsF";

    /** Path of the file from which DIAG reads the answer to display */
    public static String genDialogFile = "/space/hheat/hh/homehtStuff/genDialogF";

    /** Path of the file containing the Deep Syntactic Structure
     * XML representation of a sentence.
     * Note: this is not needed by the main DiagNLP3 module */
    public static String xmlSentenceFile = "dsynts.xml";

    /** Host where the RealProServer is running.
     * Note: this is not needed by the main DiagNLP3 module */
    public static String realProServerHost = "127.0.0.1";

    /** Port used by the RealProServer.
     * Note: this is not needed by the main DiagNLP3 module */
    public static int realProServerPort = 5556;
}
```

APPENDIX A (continued)

****
Class Utility in Package diagnlp3.misc
****

```
package diagnlp3.misc;

import java.util.*;

/**
 * This class implements some utility functions.
 * @author Davide Fossati
 */

public class Utility {
    /**
     * Splits a String of the type "a, b, c, xyz"
     * in a Vector of Strings of the type ["a", "b", "c", "xyz"].
     * @param s the String to split
     * @return a Vector of Strings
     */
    public static Vector splitString(String s) {
        Vector result = new Vector();
        int beginIndex = 0;
        int endIndex;
        int comma = ',';

        while((endIndex = s.indexOf(comma, beginIndex)) != -1) {
            String s1 = s.substring(beginIndex, endIndex).trim();
            result.add(s1);
            beginIndex = endIndex + 1;
        }

        String s1 = s.substring(beginIndex, s.length()).trim();
        result.add(s1);

        return result;
    }


    /** Builds the power set of a given Vector.
     * @param s1 the input Vector
     * @return a Set containing all the possible subsets of s1;
     */
    public static Vector buildPowerSet(Vector s1) {
        Vector result = new Vector();

        if(s1.size() <= 1) {
            result.add(new Vector(s1));
            return result;
        }

        Object f = s1.firstElement();
        Vector t1 = new Vector();
        t1.add(f);
        result.add(t1);

        Vector t2 = new Vector(s1);
        t2.remove(f);

        Vector p = Utility.buildPowerSet(t2);
        Iterator ip = p.iterator();
        while(ip.hasNext()) {
            Vector k = (Vector)ip.next();
            result.add(k);
```

APPENDIX A (continued)

```
            Vector tk = new Vector((Vector)k);
            tk.add(f);
            result.add(tk);
        }

        return result;
    }
}
```

APPENDIX A (continued)

```
      ************************************************************************
                  Class SurfaceRealizer in Package diagnlp3.real
      ************************************************************************

package diagnlp3.real;

import com.cogentex.real.api.RealProMgr;     //RealPro Manager
import org.w3c.dom.*;                        //Definition of DOM classes
import org.apache.xerces.dom.DocumentImpl;   //Implementation of DOM Document

/**
 * This is the surface realizer used by DiagNLP3.
 * It is based on the RealPro system by CoGenTex inc.
 * @author Davide Fossati
 */

public class SurfaceRealizer {
    /**
     * Default constructor
     */
    public SurfaceRealizer() {
        //Create an instance of RealPro Manager with its default configuration
        System.out.println("Initializing RealPro...");

        realproMgr = new RealProMgr();

        //Verify if the RealPro Manager has been initialized correctly.
        if (!realproMgr.isInitialized()) {
            System.out.println("Initializing RealPro ... failed; " +
realproMgr.getErrorMsg());
            System.out.println();
        }
        else {
            System.out.println("Initializing RealPro ... completed.");
            System.out.println();
        }
    }


    /**
     * Realizes a sentence from its Deep Syntactic Structure representation.
     * @param dsynts the Deep Syntactic Structure representation of the sentence
     * @return the realized sentence
     */
    public String realizeSentence(Element dsynts) {
        String realizedSentence;

        if(dsynts == null) {
            return "";
        }

        //Verify if the RealPro Manager has been initialized correctly
        if (!realproMgr.isInitialized()) {
            System.out.println("RealPro Manager not initialized; " +
realproMgr.getErrorMsg());
            return "";
        }

        //Realizes the sentence
        org.w3c.dom.DocumentFragment fragment = realproMgr.realize(dsynts);

        if (fragment.hasChildNodes()) {
            realizedSentence = realproMgr.getSentenceString();
        }
```

APPENDIX A (continued)

```
        else {
            System.out.println("Realization failed:");
            System.out.println(realproMgr.getErrorMsg());
            realizedSentence = "";
        }

        return realizedSentence;
    }


    private RealProMgr realproMgr;
}
```

APPENDIX A (continued)

```
      ************************************************************************
                    Class Adjective in Package diagnlp3.sel
      ************************************************************************

package diagnlp3.sel;

/**
 * This subclass of SyntacticElement represents an adjective.
 * @author Davide Fossati
 */
public class Adjective extends SyntacticElement {

    public Adjective(String name) {
        this.name = name;
        grammaticalCategory = "adjective";
    }
}
```

APPENDIX A (continued)

****
Class CommonNoun in Package diagnlp3.sel
****

```
package diagnlp3.sel;

/**
 * This subclass of SyntacticElement represents a common noun.
 * @author Davide Fossati
 */
public class CommonNoun extends SyntacticElement {

    public CommonNoun(String name) {
        this.name = name;
        grammaticalCategory = "common_noun";
    }

    public String getFeatures() {
        StringBuffer s = new StringBuffer();
        s.append("lexeme=" + name);
        s.append(" class=" + grammaticalCategory);
        if(number != null)
            s.append(" number=" + number);
        if(gender != null)
            s.append(" gender=" + gender);
        if(case1 != null)
            s.append(" case=" + case1);
        if(article != null)
            s.append(" article=" + article);

        return s.toString();
    }



    public String number = null;
    public String gender = null;
    public String case1 = null; // this variable is named "case1" instead of "case"
                                // because "case" is a java keyword
    public String article = null;
}
```

APPENDIX A (continued)

****
Class PersonalPronoun in Package diagnlp3.sel
****

```
package diagnlp3.sel;

/**
 * This subclass of SyntacticElement represents a personal pronoun.
 * @author Davide Fossati
 */
public class PersonalPronoun extends SyntacticElement {

    public PersonalPronoun(String person, String number) {
        this.name = "<PRONOUN>";
        grammaticalCategory = "personal_pronoun";
        this.person = person;
        this.number = number;
    }

    public String person;
    public String number;
}
```

APPENDIX A (continued)

****
Class SentenceBuilder in Package diagnlp3.sel
****

```
package diagnlp3.sel;

import diagnlp3.misc.Constants;
import diagnlp3.agg.*;
import diagnlp3.ex.*;
import diagnlp3.relrend.*;
import diagnlp3.real.*;

import java.util.*;
import java.io.*;
import java.net.*;

import org.w3c.dom.*;
import com.cogentex.textbuilder.*;
import com.cogentex.xml.*;

/**
 * This class builds intermediate level and surface level sentences
 * from higher level representations.
 * It uses the Exemplars framework by CoGenTex inc.
 * @author Davide Fossati
 */

public class SentenceBuilder {
    /**
     * Default constructor
     */
    public SentenceBuilder() {
        textBuilder = new TextBuilder();
        ExLoader.loadExemplars(textBuilder);
    }


    /**
     * Renders an Aggregation object, producing a SentenceStructure.
     * @param agg the Aggregation to be rendered
     * @return a SentenceStructure object
     */
    public SentenceStructure renderAggregation(Aggregation agg) {
        SentenceStructure ss = new SentenceStructure();
        RelationRanking rr = new RelationRanking(agg);

        SentenceStructure ss2 = new SentenceStructure();

        applyAllRelations(rr, ss);

        int l = applySubjectFromList(ss, agg.getAggregands());

        ss.validateStructure();

        return ss;
    }


    /**
     * Applies all the relations ranked in a given RelationRanking to a
     * given SentenceStructure.
     * @param rr a RelationRanking object containing all the information
     * to be rendered
     * @param ss the SentenceStructure to be modified
```

APPENDIX A (continued)

```
     */
    public void applyAllRelations(RelationRanking rr, SentenceStructure ss) {
        Vector renderers = rr.getRenderers();
        Iterator itr = renderers.iterator();
        while(itr.hasNext()) {
            RelationRenderer relRend = (RelationRenderer)itr.next();
            int l = relRend.apply(ss);
//             System.out.println("Applied relation " + relRend + ", application level
" + l);
        }
    }


    /**
     * Applies the names of the given Aggregands to the subject field of the
     * given SentenceStructure.
     * @param ss the SentenceStructure to be modified
     * @param aggregands a Vector of Aggregand objects
     * @return the level of application (in the sense of a RelationRenderer object)
     */
    public int applySubjectFromList(SentenceStructure ss, Vector aggregands) {
        int level = 0;

        CommonNoun subj = new
CommonNoun(((Aggregand)aggregands.firstElement()).getName());
        CommonNoun s1 = subj;
        int n = aggregands.size();
        for(int i = 1; i < n; i++) {
            CommonNoun s2 = new
CommonNoun(((Aggregand)aggregands.elementAt(i)).getName());
            s1.setCoordinate(s2);
            s1 = s2;
        }

        if(ss.subject == null && ss.verb == null) {
            level = 1;
            ss.subject = subj;
            ss.verb = new Verb("exist");
        }
        else if (ss.subject == null) {
            level = 2;
            ss.subject = subj;
        }
        else if (ss.verb == null) {
            level = 3;
            ss.verb = new Verb("exist");
        }

        return level;
    }


    /**
     * Renders a SentenceStructure object into a plain text realized String.
     * @param ss the SentenceStructure to be rendered
     * @param realizer the SurfaceRealizer used to realize the sentence
     * @return the realized final sentence
     */
    public String renderSentenceStructure(SentenceStructure ss, SurfaceRealizer
realizer) {
        Exemplar startExemplar = new StructureRenderer(ss);
        Document myDoc = textBuilder.buildXmlDoc(startExemplar);
        Element myElement = (Element)myDoc.getElementsByTagName("dsynts").item(0);
        String mySentence = realizer.realizeSentence(myElement);
```

APPENDIX A (continued)

```
        return mySentence;
    }


    /**
     * Renders a SentenceStructure object into a plain text realized String.
     * This method assumes that an external RealProServer is running.
     * @param ss the SentenceStructure to be rendered
     * @return the realized final sentence
     */
    public String renderSentenceStructure(SentenceStructure ss) {
        Exemplar startExemplar = new StructureRenderer(ss);
        Document myDoc = textBuilder.buildXmlDoc(startExemplar);
        String mySentence = "";

        try {
            PrintWriter out1 = new PrintWriter(new BufferedWriter(new
FileWriter(Constants.xmlSentenceFile)));
            XmlUtils.toXML(myDoc, out1);

            Socket sock = new Socket(Constants.realProServerHost,
Constants.realProServerPort);
            InputStream inStream = sock.getInputStream();
            byte[] buf = new byte[1000];
            int n = inStream.read(buf);
            mySentence = new String(buf, 0, n);
            sock.close();

        } catch (Exception ex) { ex.printStackTrace(); }

        return mySentence;
    }


    private TextBuilder textBuilder;

}
```

APPENDIX A (continued)

Class SentenceStructure in Package diagnlp3.sel

```
package diagnlp3.sel;

/**
 * This class represents a typical sentence structure, composed by a subject,
 * a subject global attribute, a pre-verbal element, a verb, an object, and an element
 * in the final position of the sentence.
 * Each element can be an insance of any subclass of SyntacticElement, except the
 * verb that must be an instance of class Verb.
 * Each element is accessible as a public variable.
 * Before the rendering, a SentenceStructure should be validated
 * with its validateStructure() method.
 * @author Davide Fossati
 */
public class SentenceStructure {
    public SentenceStructure() {
    }

    /**
     * Displays this SentenceStructure in a human readable format.
     */
    public void display() {
        System.out.println("subject: " + subject);
        System.out.println("subjectGlobalAttribute: " + subjectGlobalAttribute);
        System.out.println("preVerbal: " + preVerbal);
        System.out.println("verb: " + verb);
        System.out.println("object: " + object);
        System.out.println("sentFinal: " + sentFinal);
    }


    /**
     * Validates this SentenceStructure
     */
    public void validateStructure() {
        fixAgreement();

        if(subject != null && subjectGlobalAttribute != null) {
            subject.setGlobalAttribute(subjectGlobalAttribute);
        }
               subjectGlobalAttribute = null;
    }


    /**
     * Fixes the agreements of some elements in this SentenceStructure.
     */
    public void fixAgreement() {
        // subject - object agreement
        if(subject != null && object != null) {
            if(subject.getCoordinate() != null && object.getPreposition() == null) {
                if(object.getClass().getName().endsWith("CommonNoun")) {
                    ((CommonNoun)object).number = "pl";
                }
            }
            else if(subject.getCoordinate() == null && object.getPreposition() ==
null) {
                if(object.getClass().getName().endsWith("CommonNoun")) {
                    ((CommonNoun)object).number = "sg";
                }
            }
```

APPENDIX A (continued)

```
        }

        // subject - preVerbal agreement
        if(subject != null && preVerbal != null) {
            if(subject.getCoordinate() != null && preVerbal.getPreposition() == null)
{
                if(preVerbal.getClass().getName().endsWith("CommonNoun")) {
                    ((CommonNoun)preVerbal).number = "pl";
                }
            }
            else if(subject.getCoordinate() == null && preVerbal.getPreposition() ==
null) {
                if(preVerbal.getClass().getName().endsWith("CommonNoun")) {
                    ((CommonNoun)preVerbal).number = "sg";
                }
            }
        }

        // object - sentFinal agreement
        if(object != null && sentFinal != null) {
            if(object.getCoordinate() != null && sentFinal.getPreposition() == null) {
                if(sentFinal.getClass().getName().endsWith("CommonNoun")) {
                    ((CommonNoun)sentFinal).number = "pl";
                }
            }
            else if(object.getCoordinate() == null && sentFinal.getPreposition() ==
null) {
                if(sentFinal.getClass().getName().endsWith("CommonNoun")) {
                    ((CommonNoun)sentFinal).number = "sg";
                }
            }
        }
    }


    /** The logic subject of this sentence */
    public SyntacticElement subject = null;

    /** The global attribute of the subject of this sentence */
    public SyntacticElement subjectGlobalAttribute = null;

    /** The element in pre-verbal position of this sentence */
    public SyntacticElement preVerbal = null;

    /** The main verb of this sentence */
    public Verb verb = null;

    /** The logic object (direct or indirect) of this sentence */
    public SyntacticElement object = null;

    /** The element in final position of this sentence */
    public SyntacticElement sentFinal = null;
}
```

APPENDIX A (continued)

****
Class SyntacticElement in Package diagnlp3.sel
****

```
package diagnlp3.sel;

/**
 * This class represent a generic SyntacticElement.
 * This class will be specialized in nouns, adjectives, verbs, etc.
 * Each subclass will have a different set of specific features (variables),
 * corresponding to the features described in the RealPro system manual.
 * @author Davide Fossati
 */
public class SyntacticElement {
    /**
     * Default constructor.
     */
    public SyntacticElement() {
        grammaticalCategory = "";
    }


    /**
     * Constructor.
     * @param name the name (lexeme) of this SyntacticElement.
     */
    public SyntacticElement(String name) {
        this.name = name;
        grammaticalCategory = "";
    }


    /**
     * Returns the grammatical category (noun, verb, etc) of this object.
     * @return the grammatical category of this object.
     */
    public String getGrammaticalCategory() {
        return grammaticalCategory;
    }


    /**
     * Returns the name (lexeme) of this object.
     * @return the name of this object.
     */
    public String getName() {
        return name;
    }


    /**
     * Sets a new name (lexeme) to this object.
     * @param name the new name to be set.
     */
    public void setName(String name) {
        this.name = name;
    }


    /**
     * Returns the preposition associated to this object.
     * @return the preposition associated to this object.
     */
    public String getPreposition() {
```

APPENDIX A (continued)

```
    return preposition;
}


/**
 * Sets a new preposition to this object.
 * @param prep the new preposition to be set.
 */
public void setPreposition(String prep) {
    preposition = prep;
}


/**
 * Returns the first coordinate ("and") to this object.
 * @return the first coordinate SyntacticElement to this object.
 */
public SyntacticElement getCoordinate() {
    return coord;
}


/**
 * Returns the SyntacticElement subordinate (attribute) to this object.
 * @return the subordinate to this object.
 */
public SyntacticElement getAttribute() {
    return attr;
}


/**
 * Returns the global attribute of this object.
 * @return the global attribute of this object.
 */
public SyntacticElement getGlobalAttribute() {
    return globalAttr;
}


/**
 * Adds a coordinate SyntacticElement to the actual list
 * of coordinates.
 * @param s the SyntacticElement to be coordinated.
 */
public void addCoordinate(SyntacticElement s) {
    if(coord == null) {
        setCoordinate(s);
    }
    else {
        coord.addCoordinate(s);
    }
}


/**
 * Sets the first coordinate SyntacticElement to this object.
 * @param the new first coordinate SyntacticElement to be set.
 */
public void setCoordinate(SyntacticElement s) {
    coord = s;
    fixAgreement();
    fixArticles();
}
```

APPENDIX A (continued)

```
    /**
     * Sets the subordinate SyntacticElement (attribute) to this object.
     * @param the new attribute to be set.
     */
    public void setAttribute(SyntacticElement s) {
        attr = s;
    }


    /**
     * Sets the global attribute of this object.
     * @param the new global attribute to be set.
     */
    public void setGlobalAttribute(SyntacticElement s) {
        globalAttr = s;
        fixAgreement();
        fixArticles();
    }


    /**
     * Returns the features of this object.
     * @return a String representing the list of features of this object.
     */
    public String getFeatures() {
        return "lexeme=" + name +
            " class=" + grammaticalCategory;
    }


    public String toString() {
        return getFeatures() +
            ", GL_ATTR= (" + globalAttr +
            "), ATTR= (" + attr +
            "), COORD= (" + coord + ")";
    }


    /**
     * Fixes some agreement problems, modifying some features in
     * a suitable way.
     */
    private void fixAgreement() {
        if(coord != null && globalAttr != null) {
            if(globalAttr.getClass().getName().endsWith("CommonNoun")) {
                ((CommonNoun)globalAttr).number = "pl";
            }
        }
        else if(coord == null && globalAttr != null) {
            if(globalAttr.getClass().getName().endsWith("CommonNoun")) {
                ((CommonNoun)globalAttr).number = "sg";
            }
        }
    }


    /**
     * Fix the article feature of this object, according to agreement rules.
     */
    private void fixArticles() {
        if(globalAttr != null &&
globalAttr.getClass().getName().endsWith("CommonNoun")) {
```

APPENDIX A (continued)

```
            fixArticles(this.coord);
        }
    }


    /**
     * Internal method used by fixArticles().
     */
    private void fixArticles(SyntacticElement elem) {
        if(elem != null) {
            if(elem.getClass().getName().endsWith("CommonNoun")) {
                ((CommonNoun)elem).article = "no-art";
            }
            fixArticles(elem.coord);
        }
    }



    protected String grammaticalCategory;

    protected String name = null;
    protected String preposition = null;
    protected SyntacticElement coord = null;
    protected SyntacticElement attr = null; // used for attributes valid for this
element only
                                            // (not for its coordinates)
    protected SyntacticElement globalAttr = null; // should be used for attributes
that are global
                                                  // with respect to all the
coordinates
}
```

APPENDIX A (continued)

****
Class Verb in Package diagnlp3.sel
****

```
package diagnlp3.sel;

/**
 * This subclass of SyntacticElement represents a verb.
 * @author Davide Fossati
 */
public class Verb extends SyntacticElement {

    public Verb(String name) {
        this.name = name;
        grammaticalCategory = "verb";
    }

    public String getFeatures() {
        StringBuffer s = new StringBuffer();
        s.append("lexeme=" + name);
        s.append(" class=" + grammaticalCategory);
        if(tense != null)
            s.append(" tense=" + tense);
        if(voice != null)
            s.append(" voice=" + voice);
        if(aspect != null)
            s.append(" aspect=" + aspect);
        if(taxis != null)
            s.append(" taxis=" + taxis);
        if(mood != null)
            s.append(" mood=" + mood);
        if(polarity != null)
            s.append(" polarity=" + polarity);
        if(question != null)
            s.append(" question=" + question);

        return s.toString();
    }

    public String tense = null;
    public String voice = null;
    public String aspect = null;
    public String taxis = null;
    public String mood = null;
    public String polarity = null;
    public String question = null;
}
```

APPENDIX A (continued)

```
      ************************************************************************
       File StructureRenderer.ex (Exemplars used to build Package diagnlp3.ex)
      ************************************************************************

package diagnlp3.ex;

import diagnlp3.sel.*;

base XmlExemplar;

exemplar StructureRenderer(SentenceStructure ss) {
  void apply() {
    <<+
      <html>
      <dsynts>
          {{ RenderMainVerb(ss) }}
      </dsynts>
      </html>
    +>>
  }
}



exemplar RenderSyntacticElement(SyntacticElement elem, String relation, String
position) {
  void apply() {
    if(elem != null) {
      String preposition = elem.getPreposition();
      if(preposition != null) {
        <<+
          <dsyntnode lexeme={ preposition }
                     class="preposition"
                     rel={ relation }
                     position={ position } >

            {{ RenderSyntElemWithoutPreposition(elem, "II", "") }}

          </dsyntnode>
        +>>
      }
      else {
        <<+
          {{ RenderSyntElemWithoutPreposition(elem, relation, position) }}
        +>>
      }
    }
  }
}



exemplar RenderSyntElemWithoutPreposition(SyntacticElement elem, String relation,
String position) {
  void apply() {
    <<+
      <dsyntnode lexeme={ elem.getName() }
                 class={ elem.getGrammaticalCategory() }
                 rel={ relation }
                 position={ position } >

        {{ RenderGlobalAttribute(elem) }}
        {{ RenderAttribute(elem) }}
        {{ RenderCoordination(elem) }}
```

APPENDIX A (continued)

```
      </dsyntnode>
    +>>
  }
}


exemplar RenderCoordination(SyntacticElement s) {
  void apply() {
    SyntacticElement c1 = s.getCoordinate();
    if(c1 != null) {
      <<+
        <dsyntnode lexeme="and"
                   class="coordinating_conj"
                   rel="COORD">

          {{ RenderSyntacticElement(c1, "II", "") }}

        </dsyntnode>
      +>>
    }
  }
}


exemplar RenderAttribute(SyntacticElement s) {
  void apply() {
    SyntacticElement a1 = s.getAttribute();
    if(a1 != null) {
      <<+
        {{ RenderSyntacticElement(a1, "ATTR", "") }}
      +>>
    }
  }
}


exemplar RenderGlobalAttribute(SyntacticElement s) {
  void apply() {
    SyntacticElement a1 = s.getGlobalAttribute();
    if(a1 != null) {
      <<+
        {{ RenderSyntacticElement(a1, "ATTR", "") }}
      +>>
    }
  }
}


exemplar RenderCommonNoun(CommonNoun elem, String relation, String position) extends
RenderSyntElemWithoutPreposition {
  void apply() {
    <<+
      <dsyntnode lexeme={ elem.getName() }
                 class="common_noun"
                 rel={ relation }
                 position={ position }
                 number={ elem.number }
                 gender={ elem.gender }
                 case={ elem.case1 }
                 article={ elem.article } >

        {{ RenderGlobalAttribute(elem) }}
```

APPENDIX A (continued)

```
        {{ RenderAttribute(elem) }}
        {{ RenderCoordination(elem) }}

      </dsyntnode>
    +>>
  }
}


exemplar RenderPersonalPronoun(PersonalPronoun elem, String relation, String position)
                               extends RenderSyntElemWithoutPreposition {
  void apply() {
    <<+
      <dsyntnode lexeme={ elem.getName() }
                 class="personal_pronoun"
                 rel={ relation }
                 position={ position }
                 number={ elem.number }
                 person={ elem.person } >

        {{ RenderGlobalAttribute(elem) }}
        {{ RenderAttribute(elem) }}
        {{ RenderCoordination(elem) }}

      </dsyntnode>
    +>>
  }
}


exemplar RenderMainVerb(SentenceStructure ss) {
  void apply() {
    <<+
      <dsyntnode lexeme={ ss.verb.getName() }
                 tense={ ss.verb.tense }
                 voice={ ss.verb.voice }
                 aspect={ ss.verb.aspect }
                 taxis={ ss.verb.taxis }
                 mood={ ss.verb.mood }
                 polarity={ ss.verb.polarity }
                 question={ ss.verb.question }
                 class="verb"
                 rel="nil">

        {{ RenderGlobalAttribute(ss.verb) }}
        {{ RenderAttribute(ss.verb) }}
        {{ RenderCoordination(ss.verb) }}

          {{ RenderSyntacticElement(ss.subject, "I", "") }}
          {{ RenderSyntacticElement(ss.object, "II", "") }}
          {{ RenderSyntacticElement(ss.preVerbal, "ATTR", "pre-verbal") }}
          {{ RenderSyntacticElement(ss.sentFinal, "ATTR", "sent-final") }}

        </dsyntnode>
      +>>
    }
}
```

APPENDIX B

TABLE XVII. ATTRIBUTES TO BE USED IN "SYNTACTICELEMENT" SUBCLASSES.

SyntacticElement subclass	***Attribute***	***Value***	***Default***	***Examples***
Verb	tense	pres	y	John loves Mary
		past		John loved Mary
		future		John will love Mary
	voice	act	y	John loves Mary
		pass		John is loved by Mary
	aspect	simple	y	Mary eats an ice-cream
		cont		Mary is eating an ice-cream
	taxis	nil	y	John loves Mary
		perf		John has loved Mary
	mood	ind	y	John loves Mary
		cond		John would love Mary
		imp		Call Mary
		inf		John love Mary
		inf-to		For John to love Mary would be a problem
		pres-part		John loving Mary (is a problem)
		past-part		Given the book, Mary disappeared
	polarity	nil	y	John loves Mary
		neg		John does not love Mary
	question	-	y	John loves Mary
		+		Does John love Mary?

APPENDIX B (continued)

SyntacticElement subclass	*Attribute*	*Value*	*Default*	*Examples*
CommonNoun	number	sg	y	unit
		pl		units
	gender	masc	y	boy
		fem		waitress
		neut		piano
		dual		teacher
	case1	nom	y	unit; he
		gen		unit's; his
		obj		unit; him
	article	indef	y	a unit; units
		def		the unit; the units
		dem-prox		this unit; these units
		dem-dist		that unit; those units
		no-art		unit; units
PersonalPronoun	person	1st	y	I; we
		2nd		you
		3rd		they
	number	sg	y	I
		pl		we

VITA

NAME:	Davide Fossati
EDUCATION:	M.S., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2003.
	B.S. Equivalent, Computer Science Engineering, Politecnico di Milano, Milano, Italy, 2001.
TEACHING EXPERIENCE:	Teacher, Grade School of Gaggiano (Milano), Italy: Basic Computer Science for Grade School Children, 2003.
	Teaching Assistant, Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois: CS 421 Natural Language Processing, 2002.
	Teaching Assistant, Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois: CS 202 Algorithms and Data Structures II, 2002.
	Teaching Assistant, Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois: CS 102 Introduction to Programming, 2001.
HONORS:	Research Assistantship, Natural Language Processing Lab, University of Illinois at Chicago, Chicago, Illinois 2002.
	Teaching Assistantship, Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2001-2002.