Verbalization in the Context of a Tutoring System for Description Logics

by

Shamima Mithun

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Computer Science
Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Verbalization in the Context of a Tutoring System for Description Logics

by

Shamima Mithun

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

of

Master of Science

Abstract

Intelligent Tutoring Systems (ITS) for various domains such as high-school algebra, geometry, and programming languages [20] are becoming popular and more accepted, as they help users to improve their performance in the respective domain, develop their cognitive reasoning, and reduce the learning time in obtaining required skills [14]. More recently, ITS with natural language interfaces have been developed using natural language generation techniques, mainly for error feedback and to provide suggestions and explanations to the user. None of these systems developed so far has used the verbalization of user inputs as an additional source of learning and reinforcement.

In my thesis I consider ITS, which deal with formal languages as the domain of their application, especially formal logic or mathematics. My assumption is that these ITS could be more effective if they provided a verbalization of the user input, which is a formal expression of the application domain. Verbalization has been found useful for transforming computer generated proofs, which are normally difficult to read and to understand, into the more legible format of natural language text [15, 17]. In my thesis, I explore verbalization in the context of a tutoring system for Description Logic (DL)[25], more specifically the language provided by the PowerLoom knowledge representation system [16]. Description Logic gained support and interest in the past years due to their role in the semantic web development [9], but they are difficult to learn due to their strong formal orientation. Human tutoring is often not available, and so far, no teaching or assistant tool has been developed for any Description Logic language. As an application of DL I used PowerLoom, which is a knowledge representation system used to define and maintain knowledge base developed by the user. PowerLoom system also provides a language called PowerLoom, which is a DL dialect.

For my M. Sc. thesis, I developed a Verbalization Enhanced Tutoring System (VETS), applied to the knowledge representation language provided by the PowerLoom system.

VETS produces from the user input, which is a PowerLoom construct (some form of concept definition), a natural language sentence or text reflecting the meaning of this PowerLoom construct. The verbalization of user input through natural language text is aimed to improve the understanding of PowerLoom / DL expressions, and thus enhance user performance and satisfaction working with the PowerLoom system. Verbalization is also used to describe concepts, which are already defined and stored by the user with PowerLoom language in the knowledge base. I implemented verbalization using natural language generation techniques, particular methods from a discourse planner and surface realsizer.

Experimental results show that VETS helps users to learn the Description Logic language PowerLoom better. The effectiveness of VETS was measured in tests with potential users, in terms of the number of correct answers, the number of errors, the number of attempts taken to solve tasks, and the time taken to solve tasks, compared to users who used the PowerLoom stand alone system. Test showed there was no significant difference between two system users in terms of the number of errors and the time taken but VETS users performance was significantly better compared to PowerLoom users in terms of the number of correct answers. In VETS, the user task success rate was higher compared to the task success rate of PowerLoom user. As task difficulty increases, VETS users showed even better performance compared to the PowerLoom users. Moreover, VETS users were able to correct more of their incorrect answers compared to the PowerLoom system users. Experimental results showed that VETS users performance improved over multiple sessions in terms of the number of correct answers and the time taken. Structured questions also reflect that VETS achieved user satisfaction in terms of its usability and as a tutoring tool.

Dedication

Dedicated to my parents and my husband.

Acknowledgements

First and foremost, I am thankful to my supervisor Dr. Christel Kemke for accepting me as her Masters student and guiding me with patience, encouragement, and inspiration.

Dr. Kemke introduced me to the area of Tutoring System research and Natural Language processing.

I would like to thank Dr. Kemke and the department of Computer Science for supporting me as Research Assistant and Grader/Marker.

I would also like to thank the department of Computer Science for providing me a lab.

Special thanks to all the participants who helped me to conduct all the experiments for my thesis.

I am thankful to my thesis committee members Dr. Ralph Mason and Dr. Pourang Irani.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Intelligent tutoring systems (ITS) provide a computer-based learning environment and often use Artificial Intelligence techniques [6, 33] to provide effective learning procedures such as personalized error feedback and user adaptive task generation [14]. ITS teach fundamental concepts of application domains, as well as domain-specific problem solving techniques by assigning tasks to each user. ITS usually provide error diagnosis and correction in case of user mistakes and they give hints to guide the user in the completion of an assigned task. ITS are becoming popular and accepted as tutoring tools. Various ITS in different fields have been developed to help users in their learning.

Formal logic such as Propositional Calculus, First Order Predicate Logic, Modal Logic, and Description Logic are fundamental topics in the fields of computer science, mathematics, and philosophy. As a result, these types of logic are widely taught and practiced to improve students' cognitive learning, logical reasoning, and problem solving skills. However, students encounter difficulties in learning logic because interpreting logic formulas is a cumbersome process. They need vigorous practice to grasp the underlying ideas and problem solving techniques of logic. During practice sessions, students require feedback on their errors and suggestions for addressing their difficulties, a process which resembles human tutoring. For human tutors, it is not always possible to provide personalized tutoring and assist students according to students' schedules. As a solution to

these limitations of human tutoring, some computer-based intelligent tutoring systems have been developed in the field of logic.

ITS such as [1, 23, 30, 32] are available to teach logic. The SIAL system [32] is designed for First Order Predicate Logic. [1, 23, 30] are tutoring systems for Propositional Logic. These systems assign tasks to users in order to teach fundamental concepts and theorem proving techniques. While performing the assigned tasks, if users make mistakes, these systems diagnose users' mistakes and provide appropriate error messages and suggestions. Users often have problems in understanding logic formulas and constructs. As a result, users face difficulties in learning logic. However none of these logic tutoring systems helps to understand the meaning of logic formulas. This issue raises the need for a tutoring system which can communicate with users accompanied by logic constructs and natural language interpretation of each of those constructs besides its usual functionalities such as diagnosing user errors and providing hints. In my thesis, I develop a verbalization enhanced ITS for Description Logic.

Verbalization is defined as "Transformation of a formal expression into natural language text" [17, 15, 3]. Verbalization techniques translate logic formulas into natural language text that increases the readability and understandability of logic formulas. For example, a easy logic formula and a comparatively difficult logic formula in Description Logic and their corresponding possible verbalization are given respectively:

Logic Formula: $(defconcept\ size\ (?s)\ :<=>\ (member-of\ ?s\ (setof\ small\ big)))$

Verbalization: Size is a concept which has values small and big.

Logic Formula: $(assert\ (forall\ (?x\ ?y)(<=>\ (and(direction\ left\ ?x\ ?y)$
$(and\ (furniture\ ?x)(furniture\ ?y)))(and(direction\ right\ ?y\ ?x)$
$(and(furniture\ ?x)(furniture\ ?y))))))$

Verbalization: If x and y are two pieces of furniture, and x is left of y, then y is right of x and vice versa.

From the above examples, it is evident that verbalization makes logic formulas more readable and easy to understand. The verbalization of logic formulas gives users an

immediate feedback about whether they used the formula properly and whether the entered constructs (in general definitions of concepts to build a knowledge base) reflect what they really wanted to express. For example:

Logic Formula: (*assert* (*left_of chair table*))

Verbalization: The chair is left_of the table.

Here, maybe the user wants to place these two pieces of furniture other way round. The verbalized text will help the user to know the actual view of the knowledge base.

Verbalization has been found useful to describe computer generated proofs. Users can understand the verbalization of proofs easily, but they have difficulties in understanding computer generated proofs [3, 15, 17]. The experimental results of this study show that computer generated verbalization of proof achieved more readability and understandability compared to the computer generated proofs, which are mostly in formula format. Verbalization is also successfully used in generating weather reports [11], fairy tales [27], and for other purposes. The application of verbalization techniques in ITS for logic may bring significant improvement in areas of learning logic. This improvement motivated me to incorporate verbalization into the framework of a tutoring system for logic.

1.1 Research Outline

In my thesis, **I integrated verbalization techniques into the framework of a tutoring system for Description Logic.** The main focus of my thesis is to explore the use of verbalization techniques, with the goal to enhance the effectiveness of tutoring systems, in particular for formal logic-like languages as application domain. I have developed a verbalization enhanced tutoring system (VETS) for description logic language PowerLoom. To evaluate the effectiveness of VETS, I conducted both quantitative and qualitative studies. In quantitative experiments, I compared user performance in VETS and in the stand alone PowerLoom system in terms of the number of correct answers users provided, the number of errors users made, the number of attempts user took, the

the time taken to solve a task, the task success rate, and the task completion rate. One quantitative experiment was also conducted to measure user performance improvement over time. In the qualitative study, to measure user satisfaction, both Likert-scale and open-ended questions [26] were included in questionnaires that the users were requested to fill after their sessions using VETS.

I chose Description Logic (DL) (which are knowledge representation languages) as a concrete application to design and develop my verbalization enhanced tutoring system VETS. DL are becoming popular with the growth of the Internet because DL are used as the core language to develop the semantic webs [9]. DL are also used in various fields, including Software Engineering, Medicine, Natural Language Processing, and Web-based Information Systems [25]. Specifically, VETS is integrated with PowerLoom [16], a Description Logic system. PowerLoom is a system used to define and maintain DL knowledge base. PowerLoom knowledge representation system also provides a language called PowerLoom language that is a DL dialect. PowerLoom system is available free of charge for non-commercial applications and is the only DL language implementation that has its own user groups and mailing lists. There is no teaching or assistant tool for people who use PowerLoom or other DL languages. A tutoring system for the field of Description Logic will therefore be helpful. Using this generalized tutoring system architecture, adaptation to other logics such as Propositional Calculus and First Order Predicate Logic is possible by providing respective grammars.

With VETS, users are able to develop a knowledge base using Description Logic language PowerLoom to model their intended domain, and they are able to query the knowledge base. When the user manipulates information of the knowledge base, VETS provides two services. It analyzes the syntactic structure of the user input. If the user enters a PowerLoom construct, that is syntactically incorrect, or incompatible or inconsistent with the current knowledge base, then VETS provides error feedback and suggestions to the user. On the other hand, if the user's inputted PowerLoom construct is correct, then the system verbalizes the formula, and thus provides a natural language

interpretation of the meaning of the formula. The natural language interpretation is provided to give the user feedback on what the user is constructing and to confirm that the user's formula accurately reflects the intended expression. The verbalization is also used to provide descriptions of the knowledge base that is of currently available PowerLoom formulas that have been entered by the user. Verbalization in VETS is used for the following purposes:

1. Natural language feedback is given to the user to improve their skills and abilities regarding the use of the application language, that is their ability to use DL constructs to create, modify and access knowledge bases defined in DL in two ways

   (a) to ensure that the user's input (formal expression, logic expression, DL construct, e.g. definition of concept; assertions of facts etc.) corresponds to what the user wants to say (semantics, meaning, domain modeling, "knowledge" representation) - CONFIRMATION (of users' skills or knowledge) and

   (b) to establish and strengthen the mental association between formal DL expressions (in the computer) and informal domain descriptions (in users' minds) - CONSOLIDATION (of users' skills or knowledge).

2. Verbalization is also used to describe (parts of) the knowledge base (KB) created by the user in natural language. This feature functions to help and guide users in the development process of a KB by making users aware of the current status of the KB.

To implement error feedback and verbalization the system has four main modules: the verbalization module, the user model, the interaction handler, and the domain module. These components will be described in detail in Chapter 6. Verbalization is implemented using natural language generation techniques, in particular a Discourse Planner and a Surface Realizer.

1.2 Contributions of My Thesis

Key contributions of the system are as follows:

1. Users have difficulties understanding logic formulas, but they can understand the corresponding natural language text easily. In my thesis, I develop a verbalization method that transforms Description Logic (DL) formulas into natural language text in order to assist users to better understand and learn Description Logic formulas. The system I develop is called VETS - Verbalization Enhanced Tutoring System.

2. Description Logic languages are becoming increasingly popular not only in knowledge base development but even more due to the role of DL as a core language in the conception and standardization of the semantic web [9]. No teaching or assistant tool has been developed so far for any DL language. There is a need for a tutoring system to assist users in learning DL. VETS addresses this need.

3. Verbalization of user input can be seen as a future possibility for the improvement of Intelligent Tutoring Systems that have some kind of formal language as application domain. These application domains are formal logic (Propositional Calculus, First Order Predicate Logic, Modal Logic) but could also include other fields such as geometry, algebra, or programming languages. The verbalized formal expression, that is the natural language text corresponding to the formal expression, is intended to provide additional feedback to the user.

4. Verbalization is implemented using natural language generation (NLG) techniques. Some of the tutoring systems use NLG techniques to provide a natural language interface for error feedback and suggestions. However, they do not verbalize user input to help users to understand and learn the domain concepts. Verbalization of user input can be exploited as an additional source of learning and reinforcement, which should improve the user's overall learning process. As a result, it could enable users to work better in the application domain.

The outline of my thesis is as follows: Chapter 2 provides an introduction to tutoring system components and classifications. An overview of Description Logic is given in Chapter 3. Chapter 4 provides an introduction to the natural language generation techniques. In Chapter 5, different work related to my work is discussed. Chapter 6 is a description of the system VETS. Experimental results are summarized in Chapter 7. Chapter 8 is the discussion of the thesis. Finally, conclusions and future work are explained in chapter 9.

Chapter 2

Tutoring Systems Overview

2.1 Basic Components of a Tutoring System

Usually, an intelligent tutoring system has three main components: domain model, student model, and instruction model [10]. In addition, most of the ITS have user interfaces to communicate between the tutoring system and the user. These four modules are described in the following paragraphs.

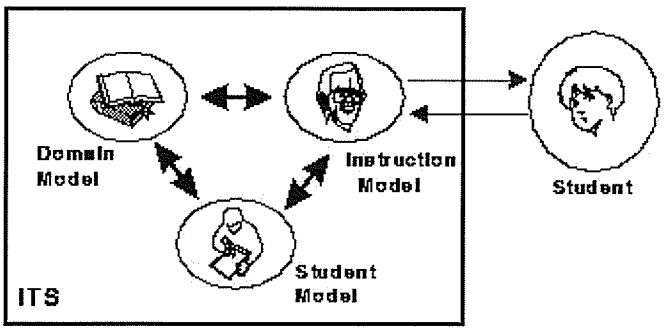

Figure 2.1: Intelligent Tutoring Systems Architecture (this figure is taken from [34]).

2.1.1 Domain Model

The domain model contains the knowledge of a specific application area [14, 34]. For example, the domain model of a tutoring system for Propositional Calculus should have the knowledge about Propositional Calculus concepts as well as the ability to understand the logic formulas generated by students and the ability to understand solved tasks such as theorem proving using Propositional Logic.

2.1.2 Student Model

"A student model represents the student's current state of knowledge, learning capabilities, background, etc" [14]. This module enables ITS to provide personalized guidance. The student model also stores information about users' misconceptions of that domain which is used to explain users' errors [4]. Misconceptions are stored as a database. Information of this module is used by the instruction model to generate user adaptive tasks, error feedback, and suggestions.

2.1.3 Instruction Model

The instruction model contains knowledge on how to teach and what to teach [14]. The instruction model retrieves information from the student model and the domain model and uses that information to accomplish the following:

- to generate tasks for users using the task generator component according to user performance and background
- to check the correctness of users' input, e.g. solutions developed by users, by comparing with the system's expected inputs, which are collected from domain experts e.g. human teachers
- to provide personalized error messages and feedback, and

- to decide whether a topic should be explained in more detail, or whether a new task should be assigned.

2.1.4 User Interface

In a tutoring system architecture such as Figure 2.1, the user interface will be included between the student and the instruction model. Different kinds of user interfaces are used in different ITS. A user interface should be simple and attractive [1]. Some systems [23, 32] use a graphical interface to represent the subject material vividly, using trees, graphs, etc. Some systems such as Logic Tutor [1] use different colors in the graphical interface. Other systems, such as LOGTUTOR [30], make use of natural language text for user interaction. Both kinds of user interface are useful to support learning.

2.2 Categorization of Tutoring Systems

Tutoring systems are categorized according to their features. Some categorizations of tutoring systems are discussed below.

2.2.1 Active Systems vs. Reactive Systems

Active systems interact with students during their sessions. These systems provide help, detect errors, and guide students to solve tasks. Active systems have detailed information about users compared to reactive systems [22]. In contrast to this, a reactive tutoring system observes user activities. In case of users' difficulties and mistakes, a reactive system comes into the foreground and provides help for the students.

2.2.2 On-site Systems vs. On-line Systems

Some tutoring systems [1, 14] are on-site systems, which do not require the web technology. Other systems [5, 23, 6] are on-line systems which are used for distant learning

using the HTTP web technology.

2.2.3 Individual Learning vs. Collaborative or Group Learning

Many systems [1, 14, 23] provide individual learning or personalized learning, whereas other systems [31] provide collaborative learning, a method where students learn by being part of a group, not necessarily from the same location. The main advantage of collaborative learning systems over individual learning systems is that in collaborative learning, students can learn from peer's misconceptions. However, in these systems students with diverse backgrounds communicate. As a result, it is very difficult to develop effective collaborative systems [31].

2.2.4 Application-dependent Systems vs. Application-independent Systems

Most of the existing tutoring systems are application dependent. Very few systems exist [6] which are application independent. These application independent systems are known as "shell systems" which are reusable. Developers can develop new systems using these shell systems as these shell systems work for diverse domains.

2.2.5 Text-based Communication vs. Dialogue-based Communication

Some systems [30] provide natural language interface to provide a more natural medium of interaction with users. Other systems [29, 28, 35] use dialogue to interact with users. Both communication types are effective in learning.

2.3 Summary

Often, ITS applications focus on selected components of the general architecture, and keep the other components simple. This idea is also followed in developing VETS system. In VETS, all the necessary components of a tutoring system are incorporated to show the functionality of a tutoring system. All these components are kept as simple as possible because my main focus is on the special aspect of verbalization. To provide the basic functionalities of a tutoring system, VETS has four main components: the User Interface, the User Model (same as the Student Model), the Domain Module, and Interaction Handler instead of the Instruction Model. VETS does not have the Instruction Model to provide tasks to users because it is developed as a reactive system. Instead, VETS has the Interaction Handler to communicate with its other modules. Moreover, VETS has a verbalization module, which is absent in typical tutoring systems.

VETS is an on-site system, which is categorized as a reactive system to give users flexibility in their work. It provides individualized learning, which is very effective in learning. In this learning environment, the tutor can provide adaptive tasks, error feedback and suggestions to the user according to the user's performance, which helps a particular user to learn the domain knowledge better. My system is an application dependent system but adaptation of other logics such as Proposition Calculus is feasible by providing the required grammar. VETS incorporates text-based interface for error feedback. In addition, it uses verbalization techniques to interpret user given DL formulas to teach Description Logic in an effective way.

Chapter 3

Knowledge Representation Using DL

3.1 Description Logic Languages

Description Logic (DL) were first introduced by Brachman et al. in [19], which describes KL-ONE (Knowledge Language One). KL-ONE and its descendants were later named "Description Logic". DL are a family of knowledge representation formalisms, which are used to represent the knowledge of an application domain as terminologies, by defining concepts and relations among concepts. Concepts are arranged in a terminological hierarchy maintaining the subconcept / superconcept relationship among them. This relation is known as subsumption relation. Assertions about the desired domains are made by creating instances of these defined concepts. A Description Logic knowledge base is comprised of two components: the "TBox" and the "ABox" [25]. The TBox (terminology box) is used to define the domain by defining concepts and relations among concepts. The ABox (assertion box) is used to specify instances of the domain, and thus to create a specific domain in accordance with the definitions in the TBox. Key features of Description Logic are their well defined semantics, as well as reasoning services, which allow systems to infer implicit knowledge from the explicit knowledge in the knowledge base and to check the consistency of concept definitions. Concepts are stored in the knowledge

base in a hierarchical manner, which makes such reasoning and inference processes faster.

DL are becoming popular as knowledge representation languages. They are now also receiving attention in the context of web languages since they are used as a knowledge representation method in the semantic web [9]. DL are used to define ontologies for the semantic web: "An ontology is a formal explicit specification of a shared conceptualization. In this context, conceptualization refers to an abstract model of some phenomenon in the domain that identifies that phenomenon's relevant concepts" [9]. There are several implemented DL systems such as RACER [25], Classic [25], PowerLoom [16], and FACT [9]. The main purpose of these knowledge representation systems is to provide high level descriptions of the domain, which can be used to build intelligent applications.

3.1.1 PowerLoom

PowerLoom is a Description Logic language used for knowledge representation. It provides a language and environment for constructing intelligent applications. PowerLoom uses Prolog [16] technology backward chainer as its deductive component for reasoning. But contrary to Prolog, PowerLoom can handle recursive rules without the risk of infinite recursions. The PowerLoom system is implemented using the programming language called STELLA [16]. The PowerLoom system is available free of charge for non-commercial applications and is the only DL language implementation that has its own user groups and mailing lists. VETS is developed for the PowerLoom language provided by the PowerLoom knowledge representation system. In VETS, the user can develop knowledge base and query the knowledge base using this language. To build and query the knowledge base VETS uses available PowerLoom application programming interface (API).

3.1.2 Knowledge Representation Using PowerLoom Constructs

A knowledge base (KB) can be developed using PowerLoom constructs with the help of TBox and ABox. An example will help to understand how we can build a KB using PowerLoom constructs. For Example we want to build a KB where we have two concepts *Color* and *Furniture*, shown in Figure 3.1. Generally a concept defines a set of similar

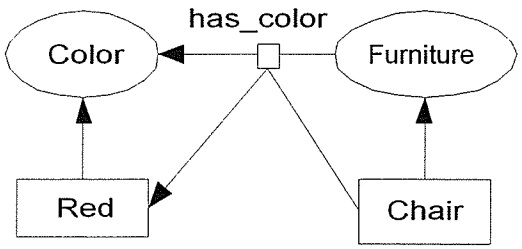

Figure 3.1: A Graphical Representation of a Sample Knowledge base

objects. We have a relation named *has_color*, which is a binary relation between *Color* and *Furniture*. These concepts and relation definitions are part of the TBox. Now with the help of the ABox we can create *Red* as an instance of *Color* and *Chair* as an instance of *Furniture*. Under the ABox we can also create the feature that *Chair has_color Red*. After creating the KB, we can also query to the KB whether *Chair has_color Red* or we can retrieve all the instances of *Furniture* and so on.

PowerLoom Constructs:

PowerLoom defines concept, relation, and function using constructs defconcept, defrelation, and deffunction. It adds and removes facts from the KB using constructs assert and retract respectively, and it finally queries the KB using ask, retrieve, and retrieve all. All these constructs are predefined keywords in PowerLoom.

TBox

Under TBox we will define concept *Furniture* and *Color* and the relation *has_color* with formulas developed using PowerLoom constructs.

Logical Formula: (*defconcept Furniture*)

Verbalization: Furniture is a concept.

Here we define the concept *Furniture* using this PowerLoom formula. Now we want to define another concept *Color* using the formula below:

Logical Formula: (*defconcept Color* (?*s*) :<=> (*member−of* ?*s* (*setof Green Red Blue*)))

Verbalization: Color has values green, red, and blue.

In the above formula ?*s* is a variable, :<=> is an operator, member-of, and setof are predefined keywords. In this formula, we are defining the concept *Color*, which can take values from the defined set of values *Green*, *Red*, and *Blue*.

Now we want to create a relation named *has_color* between the concept *Furniture* and *Color*.

Logical Formula: (*defrelation has_color* ((?*f Furniture*)(?*c Color*))))

Verbalization: has_color is a relation between furniture and color.

We have defined all our required concepts and relations under the TBox.

ABox

Now using ABox, we want to create instances of the existing concepts and relations. Here, we assign a particular *Chair* as a *Furniture* and *Red* as a *Color*.

Logical Formula:(*assert* (*Furniture Chair*))

Verbalization: Chair is a furniture.

Logical Formula: (*assert* (*Color Red*))

Verbalization: Red is a color.

Now we want to create an instance of the relation has_color.

Logical Formula: (*assert*(*has_color Chair*1 *Red*))

Verbalization: Chair1 has_color Red.

In the the ABox, we have created all the instances required to model our furniture domain.

Query to the KB

Using PowerLoom constructs, we can develop formulas to query our newly created knowledge base.

Logical Formula: $(ask(has\_color\ chair1\ Red))$

Verbalization: Ask whether chair1 has_color red.

The system will give a "true" value as an answer to this query. If this relation does not hold, the system gives a "false" value. We can do more queries using the constructs retrieve or retrieve all and we can build a more complex knowledge base using more complicated formulas. To answer queries, the PowerLoom system needs to use its inference or reasoning capabilities. When we create a new concept, the system also needs to use its reasoning to check whether the new concept is consistent with the existing concepts. For example, if we want to create a new concept that will be a subconcept of an existing concept, but the parent concept does not exist, then this new concept definition will violate the consistency of the knowledge base. As a result, the system will not allow the creation of this new concept.

Using the above mentioned PowerLoom constructs we can build a KB to model any domain. For example, we can build a KB to model the domain "university". In the university domain, *student*, *faculty*, and *class* will be concepts. *takes_class* could be a relation between the concepts *student* and *class*. Similarly, *teaches_classes* can be a relation between *faculty* and *class*. Then we can create instances of these concepts and relations under ABox, for example, a particular *student* john *takes_class CS303*. Once we develop this KB we can do different kinds of queries on it. We can make the university domain more complex by introducing the concepts of *time*, *location*, *department* and additional relevant concepts.

Chapter 4

Natural Language Generation

Natural Language Generation (NLG) is "the process of constructing natural language outputs from non-linguistic inputs" [18]. Most NLG systems have mainly two components: Discourse Planner and Surface Realizer. Apart from these two components, Microplanning is the third component that is also used for natural language generation.

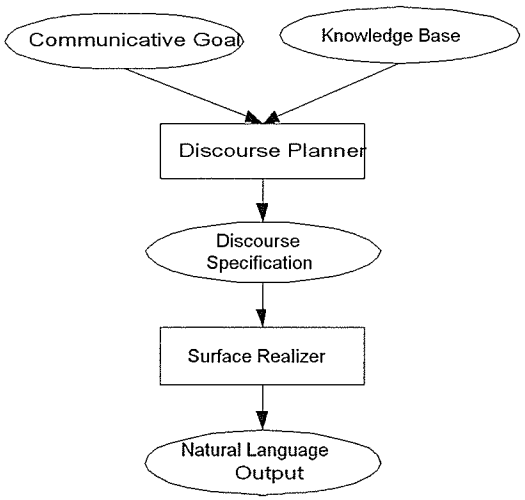

Figure 4.1: Architecture for Natural Language Generation

Discourse Planner selects content to meet a particular goal because each NLG system generates natural language output to meet an objective or goal. Discourse Planner col-

lects the content from the system knowledge base where all required information is stored. The output of Discourse Planner is called Discourse Specification. This Discourse Specification is stored in Feature Structure formats [18]. The Surface Realizer receives the Discourse Specification and generates natural language text as shown in Figure 4.1.

4.1 Discourse Planner

A Discourse Planner [18] selects contents from the knowledge base. This component selects contents which are interesting and useful in a particular context and organizes those contents into a logical form. Text Schemata and Rhetorical Relations are the typical content selection techniques [18].

Text Schemata

A Text Schemata technique requires two forms of information: one is the knowledge base of the domain and the other is the schema. The knowledge base can be represented

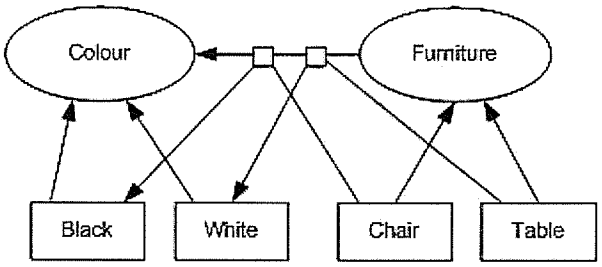

Figure 4.2: Hierarchical Knowledge base

as a hierarchical tree structure. The schema is represented as an augmented transition network (ATN). The ATN is designed according to a specific goal. For example, the goal is to verbalize the available information of the knowledge base. In the augmented network, in each of the different states, respective information will be extracted from the knowledge base. This information is passed to the Surface Realizer to generate syntactically correct sentences.

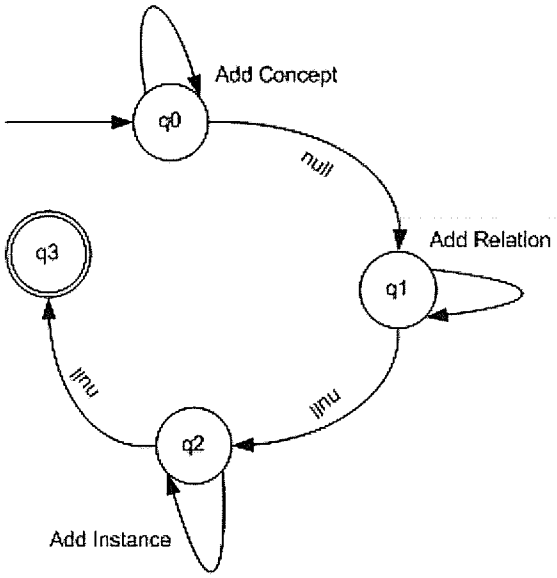

Figure 4.3: A Schema for Content Selection

I will use an example to describe the content selection procedure using Text Schemata. In this example, I will use a sample Description Logic knowledge base, which is represented as a hierarchical tree structure as shown in Figure 4.2. The schema used is represented as an augmented transition network shown in Figure 4.3, which is designed according to the goal. For this example, the goal is to verbalize the available information in the knowledge base. In Figure 4.2, *Color* and *Furniture* are concepts, and the *has_color* is the relation between these two concepts. *Black* and *White* are instances of *Color*; *Chair* and *Table* are instances of the concept *Furniture*.

In the augmented network, each of the different states will extract respective information from the knowledge base. In Figure 4.3, the state q0 will extract all concepts from the knowledge base, q1 will extract all relations, and q2 will extract all instances and reach to the final state q3. In the schema, information is stored as phrases such as *Color Concept*. These phrases are passed to the Surface Realizer as feature structure format for further processing to generate syntactically correct sentence.

Rhetorical Relations

In Rhetorical Relations, domain knowledge information is organized in a tree structure, maintaining rhetorical relations among information contents. Rhetorical relation means whether there is a contrast relation or elaboration relation between two pieces of information. For example, "I love to collect classic automobiles. My favorite car is my Rolls Royce 1902". The second sentence is an elaboration of the first sentence. "I love to collect classic automobiles. My favorite car is my 2005 Prosche". There is a contrast relation between these two pieces of information. In rhetorical structure a top-down hierarchical planner is used to extract information from this tree structured information. This planner extracts all the information necessary for a particular context and organizes it into a fixed phrase format called feature structure, which are processed by the surface realizer to generate sentences.

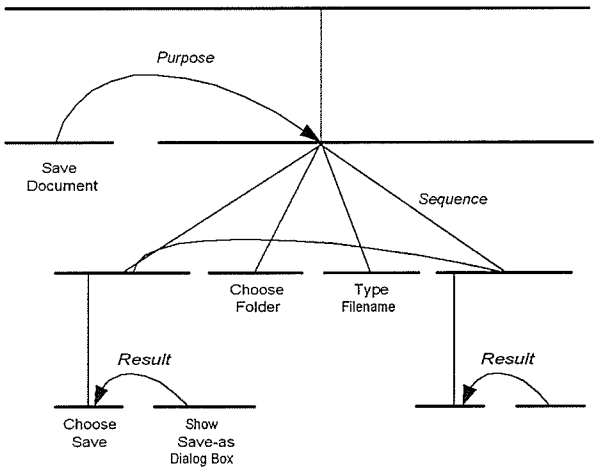

Figure 4.4: A Rhetorical Structure to Save a File (this figure is taken from [18]).

I will use an example to describe rhetorical relation techniques. This example has been taken from [18]). Here the goal is to save a document. The predefined rhetorical structure will be used to meet the goal save file. The rhetorical structure is graphically represented by the Figure 4.4. At the top level of the tree if the purpose is to save a

document then in the next level four pieces of information will be stored in a particular sequence. The first piece of information will alert users that they need to open a "Save-as dialog box" to save the file. The rest of the information will inform users that they need to choose a folder, then type a file name, and at the end they need to click the save button. Notably,a causal relationship exists between the two pieces of information, e.g. 'save document' will be the effect of clicking the save button.

4.2 Surface Realizer

A surface realizer generates syntactically correct sentence(s) from the output it receives from the discourse planner. Different techniques can be used in the surface realizer, such as Systemic Grammars, Functional Unification Grammars, or predefined Templates [18].

A Surface Realizer can use a systemic grammar to generate sentences from the input it receives from the Discourse Planner as "Feature Structures". Feature structures are a collection of syntactic and semantic features, describing the structure and contents of a sentence in a structured format, like a frame or set of records. Feature structures are elaborated in Chapter 6. The systemic grammar is represented using a directed, acyclic, and-or graph called systemic network [18]. Using the feature structure representation, the graph is traversed, and with the completion of a traversal, a sentence is generated. A surface realizer can use a Functional Unification Grammar [18] to generate sentences. In this process, feature structures are unified with the predefined functional unification grammar. This unification is done by matching the labels in the feature structures with the labels in the grammars. To unify feature structure with the functional unification grammar both should be in the same format. This process is detailed in Chapter 6.

Microplanning

Microplanning process takes place before the input passes to the surface realizer. This process decides the lexical selection, aggregation, and referring expression [18] to make

the output coherent. Lexical selection is used to select the appropriate words for that particular context; aggregation is used to bind the chunks of sentences together which are closely related; and referring expression is used to refer the used objects properly maintaining the logical connection.

4.3 Summary

In VETS, a tree based data structure is used for content selection similar to the knowledge base of the Text Schemata techniques. In VETS, there is no rhetorical relation among information. Consequently the content of VETS is not organized using rhetorical structure. Once the content of VETS is selected, the Discourse Planner creates feature structures and passes to the Surface Realizer which generates syntactically correct output for VETS. Functional Unification Grammar(FUG) is used to generate sentences. In VETS, no lexical selection component is used. In VETS, aggregation is used in the process of verbalizing the knowledge base, e.g. Corporation is a concept *if and only if* it is a sub concept of company *and* it has number of employees greater than 200. VETS also uses referring expressions to make the output coherent. One example is: "Color is a concept *which* has values red and green".

Chapter 5

Related Work

In this section, I describe tutoring systems with similar application domains as VETS, such as Logic, Mathematics, and Programming Languages, and Verbalization Systems.

5.1 Tutoring Systems with Natural Language Interface

Most of the tutoring systems use canned text for error feedback and suggestion which reduces the effectiveness of these tutoring systems. To efficiently communicate with users, some of the tutoring systems provide natural language interface using natural language generation (NLG) techniques. However, these tutoring systems do not provide verbalization which also can be achieved using NLG techniques. VETS uses NLG techniques for formula verbalization. In this section, I discuss tutoring systems which provide natural language interface using NLG techniques. These systems and my system have similarities in respect to the usability of NLG techniques even though purposes are different.

DIAG-NLP [8] is a tutoring system which teaches trouble shooting mechanisms of complex systems such as home heating and circuitry. This system is built on top of the existing tutoring system DIAG. In the original DIAG system, output was very repeti-

tive because it used predefined templates. To reduce the repetitiveness of the output of the original system, the DIAG-NLP system uses a NLG technique called Aggregation [8] to organize these templates. This system does not use NLG techniques for sentence generation. The DIAG-NLP system uses a sentence planner called EXEMPLARS [8] to generate output. EXEMPLARS selects aggregation rules to organize predefined templates. As this system uses templates for sentence generation, its output is not as coherent as that of other systems with natural language interface. To measure the effectiveness of DIAG-NLP, researchers compared the DIAG original system with DIAG-NLP system with two groups of users but they did not get statistically significant results from the experiment. DIAG-NLP resarchers also conducted qualitative measurements and found that the DIAG-NLP system won user satisfaction over the DIAG original system.

CIRCSIM [20] is a dialogue based intelligent tutoring system which assists medical students. Using this system, researchers conducted experiments to model their real domain. By analyzing student responses and tutor feedback from the experimental results, researchers developed a goal hierarchy for the CIRCSIM system. In the feedback generation process, the discourse planner determines content from this goal hierarchy to accomplish a goal, and then this information is passed to the Turn Planner [20]. The Turn Planner adds additional information to generate coherent output and generates feature structures. Then feature structures are passed to the generator, which uses the grammar formalism called Genkit [20]. The generator unifies the feature structures with the grammars and generates output. CIRCSIM researchers have analyzed 270 turns from the 5000 turns of dialogue in their collected transcripts using different hypotheses and have found that their tutoring dialog has a positive effect on learning.

The geometry tutoring system [2] is one of the tutoring systems which has a natural language interface developed by the research groups of Carnegie Mellon University. This system was developed to help students to learn through self-explanation in the context of geometry. In this system, users need to explain the steps they have taken to solve a particular problem. The system has a hierarchy of explanation categories which repre-

sents common ways of describing complete or incomplete explanations of geometric rules. The content organization of this system is similar to the goal hierarchy of the CIRCSM system. This system determines which feedback is to be given to students based on the classification of this hierarchy. Preliminary evaluation showed that performance was better for the small sample of students using the system than that of the students in the control group who did not use the tutor.

Similar to the CIRCSIM system and the geometry tutoring system, the VETS system uses a tree base data structure for content selection. VETS verbalization component selects the content for a natural language sentence according to the DL formula type. VETS uses a unification grammar for sentence generation, in the same manner as grammar is used by the CIRCSIM system. To generate interesting output, the VETS system uses grammar for sentence generation instead of templates, which are used by the DIAG-NLP system. Whereas CIRCSIM uses a turn planner to decide the content with respect to users' unexpected questions, VETS does not have any turn planner since it is not a dialog system. The main difference between VETS and the above mentioned ITS is that VETS uses NLG techniques for verbalization, which are not provided by these systems.

5.2 Logic Tutoring Systems

The Logic Tutor [1] assists computer science students to learn logic, primarily focused on teaching formal proofs. Currently, the system provides assistance with Propositional Logic. This tutor assigns adaptive tasks to users to develop their problem solving skills. To generate user adaptive tasks, the Logic Tutor sets weight to logical rules according to their difficulty levels. On the basis of users' performance and the weight of the rules, the Logic Tutor generates user adaptive tasks. While the users are in the process of solving the assigned tasks, the system provides them with feedback on errors and delivers hints and strategies to solve those tasks. To provide effective user feedback, it stores the mistake patterns in a database [1]. The Logic Tutor provides personalized help by maintaining

user usage history. In addition to all these features the modular design of the system enables it to incorporate other logics such as Predicate Calculus and First Order Predicate Logic. [1] evaluated their system using qualitative study and found that the Logic Tutor achieved users' satisfaction in terms of the usability, look of the system, and Logic Tutor as a learning tool.

P-Logic Tutor [23] is a tutoring system which teaches fundamental concepts of propositional logic and assigns students theorem proving tasks in order to teach them how to reason in logic formalism in an interactive web-based environment. During problem solving sessions, the system suggests sub-goals to users to help them achieve the primary goal. This system provides the service of a research testbed to users [23]. The research testbed is built to achieve two goals. First, it has the functionality to capture learning patterns of students. Secondly, it analyses and adaptively learns the process of human-machine interaction, thus providing an edge over other tutoring systems with the power of its interactive human-machine learning tool.

There are some similarities and differences between existing logic tutoring systems and my system VETS. Similar to the Logic Tutor, VETS can adapt to other logic such as predicate logic, propositional calculus. Similar to the logic tutor, VETS also used qualitative measurement to assess user satisfaction. On the other hand, the above-mentioned logic tutoring systems use predefined text to communicate with users, whereas VETS uses the natural language generation process to generate coherent output. The main advantage of VETS over the existing logic tutoring systems is that none of the logic tutoring systems help users to understand the meaning of logic formulas, which is vital in logic learning. The VETS system provides the meaning of logic formulas through verbalization.

5.3 Programming Tutoring Systems

5.3.1 Tutoring Systems for Logic Programming Languages

LOGTUTOR [30] is a tutoring system to teach predicate logic using the language Prolog as an application domain. In the experimental results, it was found that students learned significantly better in this system compared to in the paper pencil exercises. The LOGTUTOR interacts with users using natural language text as an effective learning method. In the system-user interaction process, the system prompts users with an English sentence that depicts a Prolog statement and also provides hints for the possible use of predefined Prolog predicates. It expects the users to write a correct Prolog statement which is equivalent to the given English statement. This system generates tasks in such a way that these tasks are related to the real world, ie. "What color is Tom's car?" so that users can semantically correlate prolog predicates with objects and relations in the real world. The system gives more attention to semantic correctness instead of syntactic correctness of the logical constructs. The main drawback of this system is that users are restricted to apply system specified Prolog predicates. The purpose of VETS is the opposite of LOGTUTOR. This is because LOGTUTOR provides natural language text that needs to be transformed into propositional logic formula, whereas VETS transforms DL formula into natural language text.

5.3.2 Tutoring Systems for Imperative Programming Languages

The Java Intelligent Tutoring System (JITS), a web based system, is designed to teach java language [33]. This system covers a subset of java language. To learn the methodologies used by students, the system models some measures such as code convention, style, and professional programming techniques to solve problems. The system uses these learned methodologies to process input given by the user. JITS has an intelligent module that detects user errors with the help of a parser and compiler, and this

module also provides suggestions. VETS similarly detects user errors, unifying with the parser's output. However, where JITS uses canned text for error feedback VETS uses NLG techniques to generate error messages.

5.4 Math Tutoring Systems

TALC [7] is a tutoring system to teach geometry. The tutor provides specifications for a geometric figure, and students construct the figure accordingly. This tutor diagnoses students' drawn geometric figures and provides correctness of the drawn figures. The main goal of this system is to improve the error explanation in cases where the students' construction is incorrect. To improve the explanation, the system uses students' misconceptions in a model of their beliefs. To represent students' knowledge, TALC uses first order predicate logic. VETS also offers explanations of users' errors.

The Object Oriented Set Tutor (OOST)[13] was developed to teach set theory to students at secondary school levels. This tutoring system was developed using the object oriented paradigm. As the system has a modular design, it can be reused and can be used in other domains. The domain knowledge of this system is organized hierarchically, which is very useful in task generation. This system is a good demonstration of the application of object orientation paradigm in the area of tutoring systems. The VETS system also follows an object-oriented architecture to adapt to other logics.

5.5 Verbalization Systems

Verbalization has been successfully used in the field of automated theorem proving, to construct natural language descriptions of automatically generated proofs, which improve the readability and understandability of these proofs. In this section, I discuss systems that use verbalization techniques for proof verbalization.

The Proverb [17] system verbalizes computer generated proofs that are similar to the

textbook proofs. The verbalized text makes the proofs readable and easily understood. The system-generated verbalized proofs are accepted by the community of automated reasoning [17]. This is the first successful system to generate logical texts from natural deduction style proofs. To generate text, the Proverb system needs to combine low-level proof steps that are not human readable to generate a high-level step which resembles human reasoning [15]. The Proverb system uses NLG techniques in its verbalization. For content selection, it uses hierarchical planning. Schema Based Planning is used for content selection [18]. Under this schema a particular predefined path is selected for a given goal. To generate syntactically correct sentences the Proverb system uses TAG-GEN [17] as a surface realizer, which uses a particular grammar formalism for this purpose. To generate smooth text, this system does some microplanning such as aggregation.

The Nuprl system [15] is designed to generate text from high level proofs. This system uses a tree structure for content selection, where each node of the tree represents one step of proof reasoning. By traversing this tree, the system selects content for verbalization according to the goal. In the Nuprl system, a linguistic component is used to map the logical concepts into words and build a sentence from those words. For sentence generation, the system uses FUF, Functional Unification Formalism, and is developed based on Functional Unification Grammar [18]. The Nuprl system is more efficient than the Proverb system because the Proverb system needs to combine low-level proof steps to generate a high-level step for verbalization. This intermediate conversion process is time consuming and needs heuristics to omit uninteresting and useless information [15], thus reducing the efficiency of the Proverb system.

The ClamNL [3] system generates natural language proofs in different forms of output according to the category of the users. As the system can generate user adaptive output, this system has advantages over the Proverb system and the Nuprl system. The system has three components: Abstraction Controller, Proof analyzer and Extractor, and Presentation planner [3]. The Abstraction Controller determines how many versions of proofs

will be generated, and it generates a detailed plan of the proofs using a tree structure. The Proof Analyzer and Extractor determines which nodes of the proof plan will be included in that particular context, based on regarding their mathematical interestingness. The Presentation Planner decides the order of the proof and maps mathematical terms into natural language. The Presentation Planner uses the EXEMPLARS framework [3] to generate text using templates. The experimental results show that ClamNL generated natural language proofs are more readable compared to computer generated proofs.

Verbalization has been successfully applied in the above-mentioned systems for verbalizing computer generated proofs. However, these systems do not contain any additional help or tutoring components. The use of verbalization in connection with tutoring systems has still to be explored. VETS provides a verbalization of a logic-like formulas, i.e. Description Logic formula / PowerLoom constructs, and integrates it with other tutoring system components into a verbalization enhanced tutoring system (for Description Logic/PowerLoom). Similar to the proof verbalization systems, VETS uses natural language generation techniques for the verbalization. VETS uses a tree structure for content selection, similar to that of the Nuprl system, and a unification grammar for sentence generation, as used by both the Proverb system and Nuprl system.

5.6 Summary

Most of the tutoring systems [1, 23] use canned text for error feedback and hints, which reduces the effectiveness of the tutoring systems. To make human-computer communication effective, some tutoring systems [8, 2, 20] provide a natural language interface using natural language generation techniques. However, none of these tutoring systems verbalize user input as additional feedback to reinforce. For ITS [1, 7] that deal with formal language domains such as logic and mathematics, verbalization of user input will be very useful because users have difficulties dealing with formal structures of the domain such as formulas. Verbalization has already been found useful in transforming computer

generated proofs into natural language text [3, 15, 17]. To make logic tutoring systems more effective, VETS incorporates a verbalization component in a tutoring system for Description Logic.

Chapter 6

VETS Description

The Verbalization Enhanced Tutoring System (VETS) is built to tutor Description Logic. Using VETS, users are able to build and query the knowledge base to model their intended domain using the Description Logic language PowerLoom. During a user session, the system verbalizes correct PowerLoom formula given as input by the user. If users provide an incorrect PowerLoom formula, the system diagnoses the error and provides an error feedback. To give the user a complete overview of the current session, the system also verbalizes the DL knowledge base, which was constructed from the user input. VETS provides these functionalities in accordance with the typical architecture of tutoring systems [10, 14, 22]. VETS consists of the following components:

- the User Interface,
- the Interaction Handler,
- the Domain Module,
- the Verbalization Module, and
- the User Model.

In the remainder of the Chapter, I describe the functionalities of each component of VETS and the methodologies it uses.

6.1 System Architecture

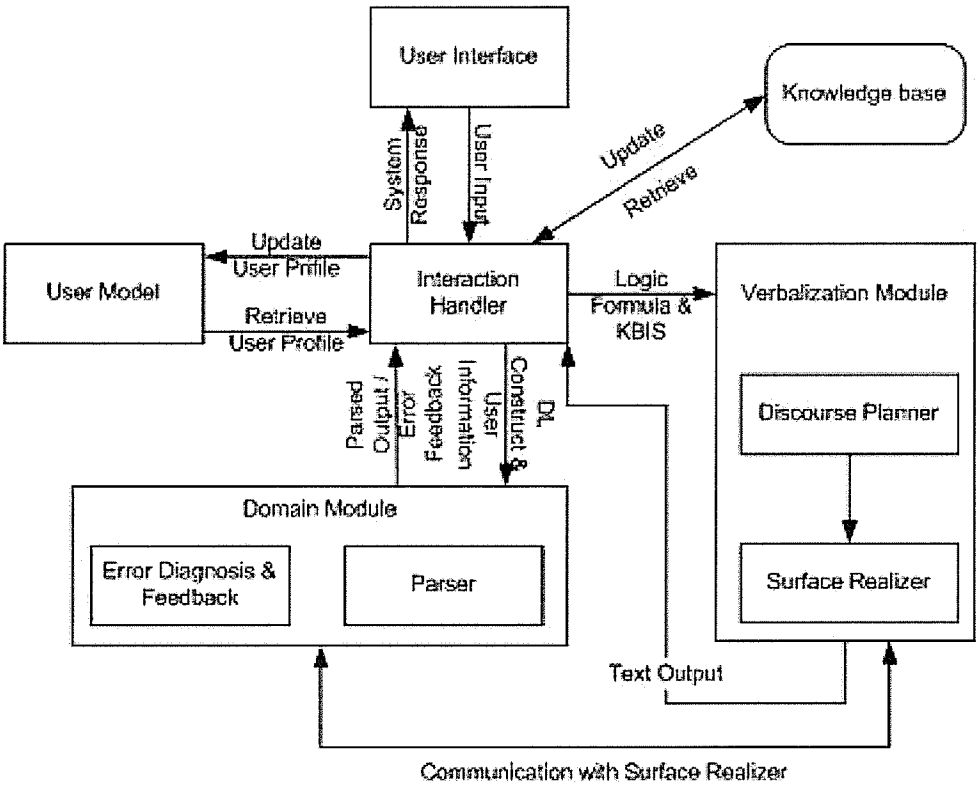

Figure 6.1: VETS Architecture

Users interact with VETS through the user interface. The interaction handler (IH) (Section 6.3) receives user input from the user interface and passes it to the domain module (DM) (Section 6.4) for diagnosis. The domain module parses the input, using a predefined grammar, which has been developed to parse the syntax of PowerLoom constructs. If the user input is correct, the domain module sends it to the interaction handler module. Then, the interaction handler module passes the parsed input to the verbalization

module (VM) (Section 6.5). The verbalization module generates a natural language text reflecting the meaning of the input DL formula and sends it to the interaction handler module. The interaction handler passes this text to the user interface which displays it on the screen. If the user input is syntactically incorrect, the error diagnosis and feedback component sends the parsed input to the surface realizer of the verbalization module, which generates an error message. The surface realizer passes the messages to the interaction handler. VETS also maintains a copy of this knowledge base to verbalize the whole knowledge base to give users overview of their sessions. VETS has a user model to generate user adaptive error feedback. This module is fully equipped with all necessary components but yet not implemented. To build and query the knowledge base VETS sends user input to the PowerLoom system.

6.2 User Interface Module

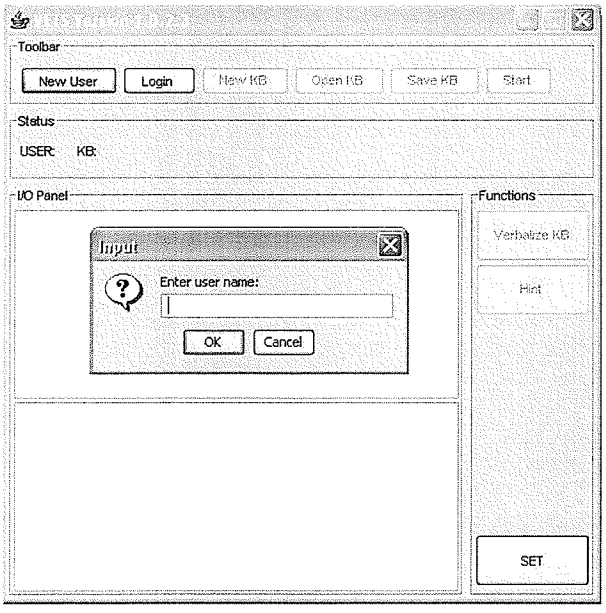

Figure 6.2: VETS User Interface

The user interacts with the VETS system through the User Interface Module (UIM). The UIM conveys all the events in the form of menu selection, button click, or text entry in a text field to the interaction handler module and displays the output as returned by the interaction handler. The user interface of VETS is shown in Figure 6.2. It consists of a toolbar with buttons to create a new user and to login an existing user. Under the toolbar, the VETS interface has buttons to create a new KB, to open a KB, and to save a KB. The user interface also has a Function toolbar, with a "verbalize KB" button to verbalize the knowledge base, a "Hint" button for help, and a "SET" button to submit user input to the system.

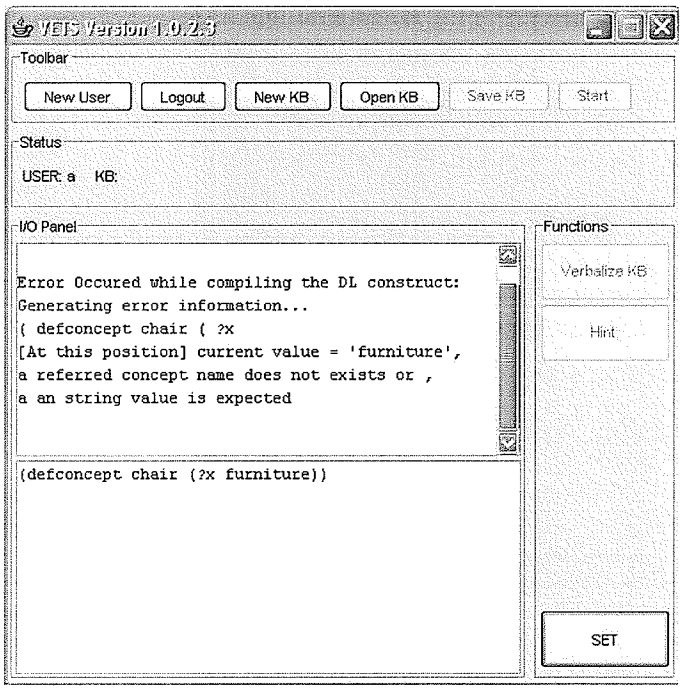

Figure 6.3: VETS I/O Panel

The interface has an I/O panel to communicate with the user. The lower part of the I/O panel is for user input and the upper part for VETS responses. Figure 6.3 shows the I/O panel of VETS with an example of the system and the user communication. The status bar of the VETS user interface in which it shows the current user name and the name

of the knowledge base created by the user. The "start" button in the toolbar starts a new task. This start button was included with the interface only to conduct evaluation experiments on the system.

6.3 Interaction Handler

The Interaction Handler (IH) communicates with other VETS components to process user input. It maintains a data structure to store information about the knowledge base built by the user. To process DL formulas, the interaction handler invokes the domain module to parse the input. In case of an error, the domain module provides error feedback to the user via the IH. If the input is correct, the domain module passes the parsed DL formula to the verbalization module. Upon receiving the parsed DL formula, the verbalization module verbalizes it and provides the verbalized output to the user. The IH also uses services of the verbalization module to verbalize the current knowledge base and articulate any concept description stored in the knowledge base in natural language text.

The IH also maintains and manages a copy of the knowledge base built by the user. The IH uses a data structure to store this copy of the knowledge base, which is used by the verbalization module for discourse planning. I call this data structure a Knowledge Base Information Store (KBIS). This copy is used by the verbalization module to verbalize the whole knowledge base.

Knowledge base Information Store

The Knowledge Base Information Store (KBIS) keeps the information in a hierarchical schema. A node in the hierarchy represents a DL concept (concept_node) or instance of a concept (instance_node). It has a special node "root_node", which is the root of the hierarchy. In Figure 6.4, *Color*, *Furniture* and *Movable_Furniture* are concept nodes whereas *Red* and *Chair* are instance nodes. The link between two concept nodes is either

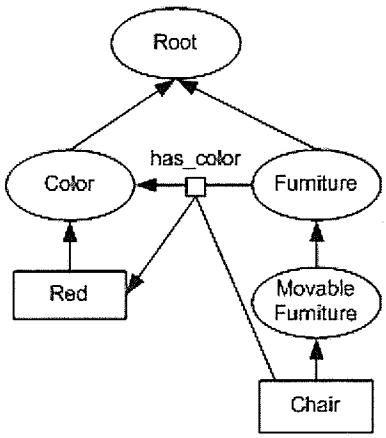

Figure 6.4: A Graphical Representation of the Sample Knowledge base

a super concept/sub concept relation or a defined DL relation between two concepts. In Figure 6.4, the link between *Furniture* and *Movable_furniture* is a super concept/sub concept relation and the link between *Color* and *Furniture* is a DL relation link.

In general, DL concept C1 will have a child node C2 if C2 is defined as a sub concept of C1, which also implies that C1 is the super concept of C2. If a new concept *Cnew* is created (using the DL defconcept construct), which has no super concept in the current hierarchy, then it will be added as child of the root node. This was done in Figure 6.4 for *Furniture*. Otherwise, *Cnew* will be added as a child node to the existing concept node of its super concept, as done in case of *Movable_Furniture* in Figure 6.4.

If a new instance is created using the DL construct *assert*, it will be added as an instance node to the corresponding concept node, as was done in Figure 6.4 in case of *Red* as an instance of *Color*. When a DL relation is defined by the user (using the DL defrelation construct), the IH creates a link between the concept nodes involved in the relation. The *has_color* is such a relation link between the concepts *Color* and *Furniture*. In this way the IH builds the KB.

6.4 Domain Module

The Domain Module (DM) maintains domain-specific information about the DL knowledge representation language, which is required and used by other components of the VETS system. The following information is stored in the DM:

- The Syntactic Grammar of the DL Language (used by the domain module)
- The Content Type Tree (used by the verbalization module)

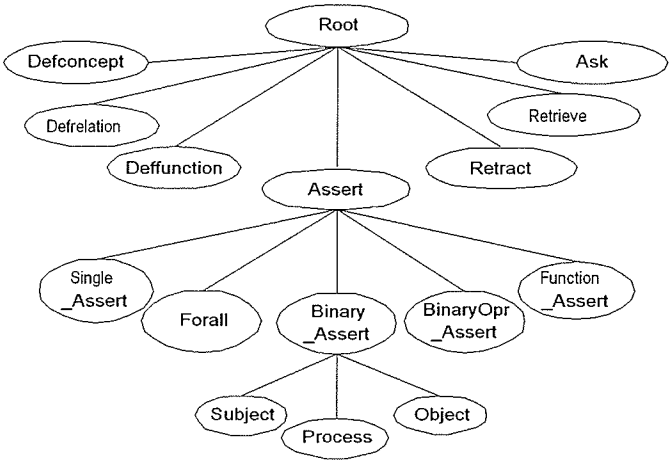

Figure 6.5: Content Type Tree

The Syntactic Grammar for the DL Language The grammar is necessary to parse DL constructs. The grammar is used by the parser to parse the user input (see Section 6.4.1).

The Content Type Tree This tree contains information about the different types of Description Logic formulas. The tree has a root node at level 0. Its children are all the DL constructs used for knowledge base manipulation (e.g. defconcept, assert) at level 1. At level 2, all the different forms of formulas that belong to each of the level 1 constructs are defined, e.g., a Single_assert or the forall constructs in Figure 6.5.

The nodes in level 2 have the necessary content information about the respective formula for verbalization. To select the content of a formula, a matched path of the tree is followed. All the information of that path will be used as content. A content type tree is shown in Figure 6.5 focusing on *binary_assert* which can be constructed using the DL construct *assert*. If we want to extract the content for a *binary_assert* type formula, we will select the following path: *assert* then *binary_assert* and then all the leaf nodes subject, process, and object. In Figure 6.5, other expansions for different DL constructs such as defconcept, defrelation, deffunction, ask, retrieve, and retract are not shown as well as of other branches of assert construct. This tree information is used by the discourse planner of the verbalization module for content selection. Content type trees for all DL formulas that the system handles are designed and stored in the domain module.

Besides handling this information, the domain module also consists of two components: the parser and the error diagnosis and feedback component.

6.4.1 Parser

The parser of the domain module uses the syntactic grammar of PowerLoom, and parses all input constructs. In VETS, a top-down parser was used. This module is integrated with the error feedback generation module. The parser analyzes the input constructs and separates each construct of the input formula. Each of the separated constructs is called 'token". For example, if the parser receives the DL formula (*defconcept chair*(?*x furniture*)), then it will analyze this formula and make eight tokens:

1. (2. defconcept 3. chair 4. (5. ?x 6. furniture 7.) 8.)

After creating these tokens the parser assigns "tag" to each token. For this example, tags are:

(	*opening parenthesis*
)	*closing parenthesis*

defconcept	*keyword*
chair, furniture	*concept name*
?x	*variable*

After assigning tags, the parser parses the input using the predefined PowerLoom grammar, and to check whether the formula is syntactically correct. If the formula is correct, the parser passes tags to the interaction handler which sends theses tags to the verbalization module to generate text corresponding to the input formula. If the formula is incorrect, the parser passes tags to the Error Diagnosis and Feedback component, which generates an error feedback.

6.4.2 Error Diagnosis and Feedback

The Error feedback Generator provides appropriate error messages for incorrect DL formulas provided as input by the user. It can send two types of error messages.

The first type addresses syntactic errors. For example, if the input formula is:

defconcept furniture)

The error message will be:

Error Message: Error Occured while compiling the DL construct.

Generating error information ...

[At this position] current value = 'defconcept', a (is expected before this

The system recognizes that in the formula an opening parenthesis is missing.

The second type of error are semantic errors. For example,

Logic Formula: (defconcept movable_furniture (?f table))

The error message will be:

Error Message: Error occurred while compiling the DL construct.

Generating error information ...

(defconcept movable_furniture(?f [At this position] current value = 'table', the

referred concept name does not exist.

Here, the user is referring to the concept name *table* which s/he did not define earlier. The system recognizes that the concept name referred to does not exist. Some sample error feedback messages are included in Appendix B.

To generate error messages, the error feedback generator provides the tags of the formula to the surface realizer (a sub module of the verbalization module) as feature structures, containing error type and error location in the formula. The surface realizer generates error messages and passes these messages to the interaction handler.

6.5 Verbalization Module

The Verbalization Module (VM) provides two services. The first service, the *formula verbalization*, transforms a correct DL input formula into a natural language statement carrying the same meaning. The second service is called *knowledge base verbalization* that means verbalization of the entire user created DL knowledge base. The verbalization module uses natural language generation techniques to achieve the goals of both of its services. In the process of verbalization, there are two distinct tasks: a) select appropriate contents and b) generate a syntactically correct, well-formed natural language sentence. For content selection, the VM uses the natural language generation component a Discourse Planner and for syntactic sentence construction it uses a Surface Realizer.

6.5.1 Formula Verbalization

VETS verbalizes simple DL formulas, as well as complex DL formulas. Verbalized text is supposed to be helpful for users in case of simple and reasonably complicated Description Logic formulas. Some VETS generated sample verbalization output is attached in Appendix A. The verbalization of DL formulas uses data from three sources: the parsed input formula with tags from the IH; the knowledge base information maintained by the

IH; and the Content Type Tree from the DM.

Discourse Planner

In the verbalization process, the discourse planner selects the contents using the above mentioned three data sources. The tags of parsed formula provide the necessary information for the content selection procedure, to determine the path to follow in the Content Type Tree. The tags are compared with the node value of the Content Type Tree, and if there is a match, then it proceeds further down the tree until it reaches a leaf node and gathers information on content requirement. After this, the discourse planner uses this content information to obtain the respective content from the parsed formula and knowledge base information. Finally, the discourse planner creates a feature structure. For each type of DL formula specific feature structures are predefined and stored.

A feature structure consists of a list of tuple(s). A tuple is of the form $< label, value >$, where label is an identifier of the information contained in the value, and value can be any string value for the label or a list of tuple(s). A label used in the feature structure corresponds to a label in the unification grammar used by the surface realizer. In the DL formula:

Input DL formula: $(assert(has\_color\ chair1\ red))$

Chair1 is a concept instance, red is another concept instance and has_color is the relation between these two instances chair1 and color. The parser will tag this formula as binary_assert. Then the discourse planner will match tags of this DL formula with the content type tree and follow the binary_assert path of assert. After extracting the content from the content type tree the discourse planner will create the corresponding feature structure:

$< CAT, ASSERT >$

$< CAT, BINARY\_ASSERT >$

$< SUBJECT, CHAIR1 >$

$< PROCESS, HAS\_COLOR >$

$< OBJECT, RED >$

This feature structure is passed to the surface realizer. The Surface realizer uses the feature structure and the unification grammar to generate the natural language output.

Surface Realizer

The Surface realizer uses the unification grammar to generate the natural language output using the feature structure. A sample unification grammar is given in Figure 6.6. The sample unification grammar has many levels, where a level is denoted with curly and square brackets. A curly bracket means that some alternative grammar rules are available to pick from at that level. Each of these alternative grammar rules is denoted with square brackets. Within a square bracket a rule is defined with a set of labels. Either an absolute value or another grammar rule, represented in square brackets, can be assigned to each of these label values. I will describe how the above mentioned feature structure will be unified with the unification grammar in Figure 6.6 to generate the sentence: Chair1 has_color red.

First, the surface realizer will find a match with the category assert because both the feature structure and the unification grammar have category assert. At this level the unification grammar has three alternative grammar rules to select such as *SINGLE_ASSERT*, *BINARY_ASSERT*, and *FUNCTION_ASST* under assert. Now, after a search, a match is found with the second alternative BINARY_ASSERT. At this stage, all the labels (SUBJECT, PROCESS, and OBJECT) of the feature structure will be matched with the labels of unification grammar. Then the unification grammar will receive required lexicon values from the feature structure and update the grammar. The updated grammar is shown in Figure 6.7.

This process continues until there are no more rules to follow. Once all the required information are generated, these information is arranged according to value of the PAT-

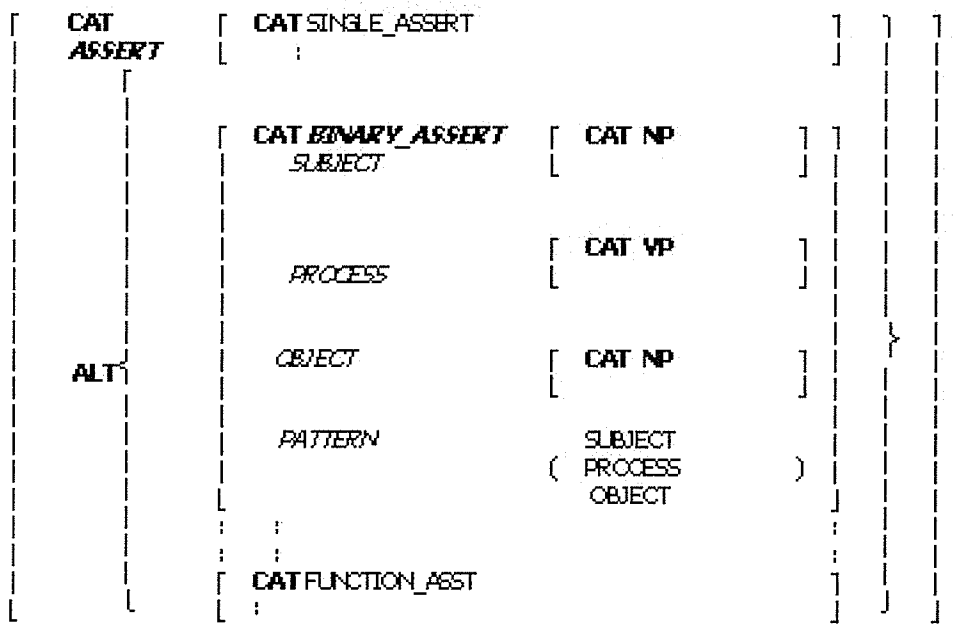

Figure 6.6: Unification Grammars

TERN label, and an output sentence is produced. In Figure 6.7 the pattern is SUBJECT PROCESS OBJECT where the lexical values for subject is chair1, process is has_color, and object is red. According the pattern these lexical values will concate and generate "chair1 has_color red". For other sentences the unification grammar needs to handle determinant, subject-verb agreement and other microplanning issues. The expansions of category NP(noun phrase), VP(verb phrase) are not shown in this figure. The surface realizer also generates error messages by unifying feature structures provided by the Error diagnosis component with this unification grammar.

6.5.2 Knowledge base Verbalization

The knowledge base verbalization service describes the current user created DL knowledge base using the knowledge base information store (KBIS) maintained by the IH. The service first gathers information for all the concepts and related information such as

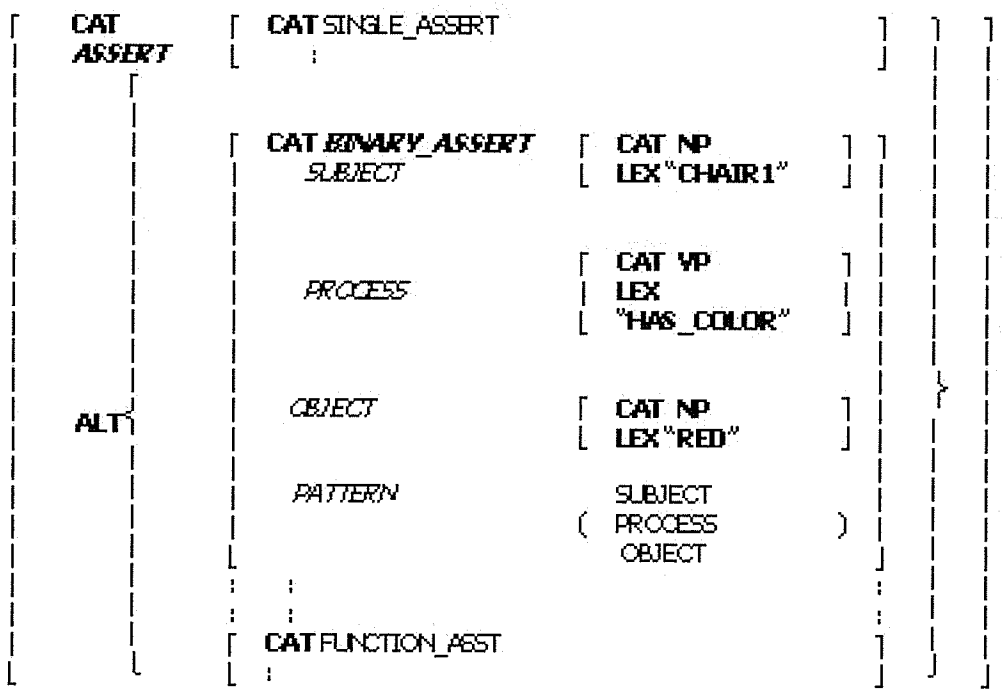

Figure 6.7: Updated Unification Grammars

relations and instances from the KBIS. For each concept, information is acquired from the KBIS, and a feature structure is created. In the case of formula verbalization, the parsed DL formulas have the tagged information, and they are used to create feature structures. In this case the tag information will be gathered for the nodes of the KBIS. Then the feature structure will be passed to the surface realizer to generate the natural language output text. The processing of the surface realizer is the same as described in the previous section. The output of surface realizer will be archived. Once all the concepts in the KBIS are verbalized the texts in the archive are displayed to the users, describing the current state of the user-created DL knowledge base. Here is a sample of KB verbalization output:

If a user entered the following DL formulas in a KB:

(*defconcept furniture*)

$(defconcept\ chair(?x\ furniture))$

$(assert(furniture\ chair1))$

$(assert(size\ small))$

$(defconcept\ size\ (?s)\ :<=>\ (member-of\ ?s\ (setof\ small\ meduim\ big)))$

$(defrelation\ has\_size(?x\ furniture)(?y\ size))$

$(assert(has\_size\ chair1\ small))$

Then the Verbalization of the KB would be:

"Furniture is a concept which has the sub-concept chair. Furniture has an instance chair1. Size is another concept which has values small, medium, and big. Has-size is a relation between furniture and size. Chair1 has_size small."

6.6 User Model

VETS is equipped to implement the user model but not yet implemented. The User Model (UM) will deal primarily with user profiles. To maintain the user profiles, this component will categorize users into two groups beginner and expert. The basic information will be maintained in a user profile be: login information, which will include username and password; user category, which will tell the assigned category to which the user will belong; and a score, which will keep track of the performance of the user. In the User Interface Module we have seen that to use the system, users need to login or create a new user. When creating a new user, if the user name already exists, the system gives the message "user name exists". During the login step, if the user name and password do not match, then the system gives the appropriate error message "invalid login". This login process is followed to maintain the user profile in future. The score of the user will be updated while the user will build the DL knowledge base. The new score will be calculated based on the number of corrections and errors the user will make during a session. The category to which a user will belong, will also be determined based on this score. A range of score values will be predefined for each category. This user model

needs to be maintained to generate user adaptive error feedback.

6.7 Walk Through Using an Example

If the user-given DL formula is:

$$(assert(has\_color\ chair1\ red))$$

then this formula will be passed to the interaction handler from the user interface. Once the interaction handler received the formula it will pass it to the domain module. The parser of the domain module will tokenize the formula and give tags to each token. Tags for this formula is shown in Section 6.4 under Parser. After giving tags, the parser will parse the formula using predefined grammar stored in the domain module. As the formula is syntactically correct the domain module will pass these tags to the interaction handler. In the next phase, the interaction handler will pass these tags to the verbalization module for verbalization. The discourse planner of the verbalization module will use these tags and content type tree information (shown in Section 6.4) which is maintained in the domain module to generate feature structure. Feature structure for the above DL formula is shown in Section 6.5 under the discourse planer. This feature structure is passed to the surface realizer. The surface realizer will unify the feature structure with the unification grammar (shown in Section 6.5) and generate syntactically correct sentence: "Chair1 has_color red". This process is shown in the Flow Diagram 6.8.

6.8 Implementation Details

VETS is developed using Java programming language. Java is a platform independent language but VEST was tested only on Windows XP operating system. Twenty six classes have been implemented to achieve different functionalities of VETS.

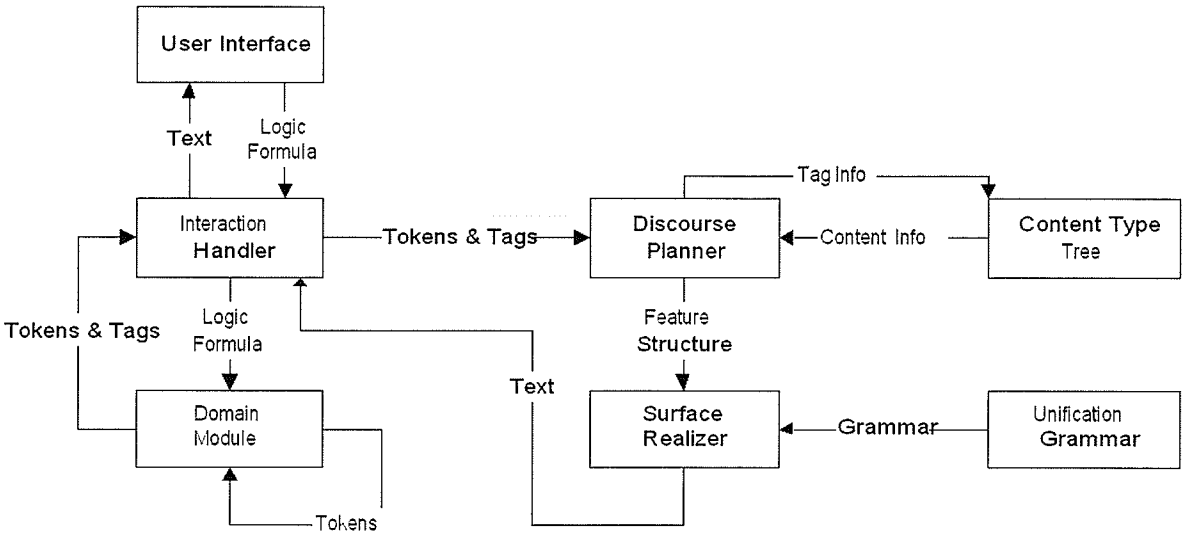

Figure 6.8: Flow Diagram for Verbalization

Integration with PowerLoom

VETS is incorporated with the PowerLoom API. To build and query the knowledge base, VETS uses PowerLoom reasoning mechanism. VETS passes user input to the PowerLoom system (running at the background using the PowerLoom API [16]), which builds the knowledge base and gives results to the queries. VETS needs to maintain a copy of this knowledge base to verbalize the whole knowledge base afterward. In VETS, a class is developed to integrate with the PowerLoom system. Web available STELLA java library is used to communicate between Java-based interface (VETS) and PowerLoom system java API.

Implemented PowerLoom Constructs

Currently, VETS can handle useful formulas generated by the seven most important DL constructs such as defconcept, defrelation, deffunction, assert, ask, retrieve, and retract. All together VETS handles twenty two different PowerLoom formulas generated by the above mentioned constructs. I did not cover other types of constructs because those are

not very frequently used. It is possible for VETS to handle other types of DL constructs by extending the PowerLoom grammar.

PowerLoom Grammar

As available PowerLoom grammar is incomplete I have developed the grammar to parse formulas which are covered by VETS. These grammar rules handle the recursive rules. These grammars are stored in a file and can be updated as required. During initialization the parser loads the grammar from the file in a dynamic data structure and uses that to parse user input.

Extension of VETS for Evaluation Process

Few additional features were added to VETS to conduct experiments. To keep track of the task which the user is performing currently a "start button" was added to the user interface. A log file for a user session where the user login information, task number, the number of correct answers, the number of attempts, the number of errors, and time taken were stored. This file was analyzed manually afterwards to measure the effectiveness of VETS.

Chapter 7

Experimental Results

Quantitative methods can be used to measure user performance as suggested by various researchers [21, 24]. Field tests are used in these methods, which examine system use in actual settings with real users. These tests enable researchers to expand the focus of an evaluation and gather information about possible unanticipated outcomes.

The performance of VETS was measured using evaluations and tests with potential users of PowerLoom and VETS, respectively. The main goal of these evaluations was to assess the effectiveness of VETS in terms of performance and the satisfaction of users. This effectiveness of VETS was measured in comparison with the stand alone PowerLoom system. Several experiments were designed, in which a set of tasks related to building a knowledge base with PowerLoom language were given to the participants of the evaluative study (see Appendix C, D, and E). Participants were mostly computer science graduate students. The participants did these tasks immediately in scheduled sessions, in open, public computer science labs. Several measurements were taken during these sessions, including the number of correct answers, the number of errors, the number of attempts taken to solve a problem (or "perform a task"), and the time taken to do a task. These were the only data obtained from the participants and these data were stored anonymously, without information about the participant. In addition, qualitative

evaluations to measure user satisfaction were obtained through structured questionnaires (see Appendix F).

Research questions, hypothesis, experimental setup, materials of the experiments, background of the participants, the design and the procedure of the experiments are described below, followed by results.

Research Questions

The research questions addressed for statistical analysis in the study were as follows.

1. Do users perform better in VETS compared to PowerLoom?
2. Is the performance consistent across different group (within subjects, between subjects) designing?
3. Does the performance vary with different task categories ("difficulty grade")?
4. Is there any performance improvement over multiple sessions?
5. What are user opinions about the contribution of verbalization in terms of their performance?

Field Pilot Test

Field testing was used to evaluate VETS in an actual experimental setting with participants. For field testing, three experiments were used.

7.1 Experiment 1

Experiment 1 was designed by following a between subjects experiment design [12] where two groups of participants perform same tasks in two different conditions where their performance is measured and compared in terms of some dependent variables. In this experiment, in order to evaluate its effectiveness, VETS was compared with the PowerLoom system, which has the same features as VETS except the verbalization. To measure the

performance of these two systems the dependent variables, the number of correct answers users gave, the number of errors users made, the number of attempts users took, and the time taken to solve tasks. In this experiment, using VETS or not using VETS (using PowerLoom) were the two conditions.

7.1.1 Hypothesis

In accordance with the ITS evaluation process [29] which uses a control group design it was expected that

1. Participants would be able to give more correct answers in the VETS system compared to the control group who used PowerLoom system.

2. Participants using VETS would make fewer errors compared to the PowerLoom users.

3. Participants who used VETS would need fewer attempts to solve tasks compared to the PowerLoom users.

4. VETS participants would need less time to solve tasks compared to the PowerLoom system participants.

7.1.2 Materials

In Experiment 1, the system VETS and the stand alone PowerLoom system, which is freely available on the web, were used. This experiment took place at a computer science graduate lab at University of Manitoba.

7.1.3 Participants

Twelve graduate students from the Computer Science Department voluntarily participated in this experiment. Two of these students had prior knowledge of Description

Logic.

7.1.4 Design

Table 7.1 illustrates the design of the experiment.

Table 7.1: Design for Experiment 1

Time	**Control Condition 6 students**	**Experimental Condition 6 students**
10 minutes	Introduction to Description Logic	
5 minutes	Demonstration of both systems	
Control: average 32 minutes Experiment: average 40 minutes	Used PowerLoom System	Used VETS System

7.1.5 Procedure

In Experiment 1, 6 students used the PowerLoom system and the other 6 students used VETS. Two students who had prior knowledge of Description Logic were assigned one to each group. The other 10 people were randomly assigned in one of the two groups. For both groups, an introduction to Description Logic was given for 10 minutes followed by a 5 minutes demonstration of respective systems where participants practiced some sample tasks with their assigned systems. For the sample tasks, answers were provided on the back side of the question paper. After the practice session, users completed the evaluation, taking 40 minutes on average. In this experiment there were no time constraints. (The questions for this experiment are attached in Appendix C). Tasks were categorized into 3 levels. Tasks 1 to 6 were categorized as difficulty level 1 tasks;

tasks 8 to 10 and 14 were categorized as difficulty level 2 tasks; and tasks 7, 11 to 13, 15 to 17 were categorized as difficulty level 3 tasks. These different categories reflect the complexity of the solution formula to be constructed as a correct answer. For higher category tasks, more complex DL formulas need to be constructed. User performance in terms of the number of correct answers, the number of errors, the number of attempts taken to solve a task, and the time taken to solve tasks for each user were recorded in separate log files (a sample log file is attached in Appendix G).

7.1.6 Results

All the data from the log files was tabulated, compared quantitatively, analyzed, and validated using the appropriate statistical tests. The Table 7.2 shows the statistical test outcome for this experiment.

Table 7.2: Statistical Test Results for Experiment 1

Dependent Variable	t-**value**	p **value**	**Critical value**	**Significant**
No. of correct answers	2.607	$p < 0.025$	2.228	YES
No. of errors	-1.733			NO
No. of attempts	-2.577	$p < 0.025$	2.228	YES
No. of tasks not attempted	2.058	$p < 0.05$	1.812	YES
Time taken	-1.333			NO
This statistical evaluation is done using t-Test (unrelated) with degrees of freedom 10 and an $\alpha = 0.05$				

Number of Correct Answers

I used the *task success rate* to evaluate the effectiveness of VETS. The task success rate

is defined as:

$$task\ success\ rate = \frac{total\ number\ of\ correct\ answers}{total\ number\ of\ tasks} \times 100$$

A higher task success rate indicates users are able to produce more correct answers in a given session. In VETS users' overall task success rate was 62%. In the PowerLoom system, the task success rate was 41% (shown in Table 7.3).

Table 7.3: Task Success Rate in Experiment 1

System Name	Number of Correct	Task Success Rate
VETS	64	62%
PowerLoom	42	41%

The average number of correct answers for each user in VETS was 11 and in PowerLoom was 7. In VETS, the average number of correct answers was 1.6 times higher than in the PowerLoom system. A comparison of the number of users completed each task in both systems is shown in Figure 7.1. Table 7.2 shows that the two systems are

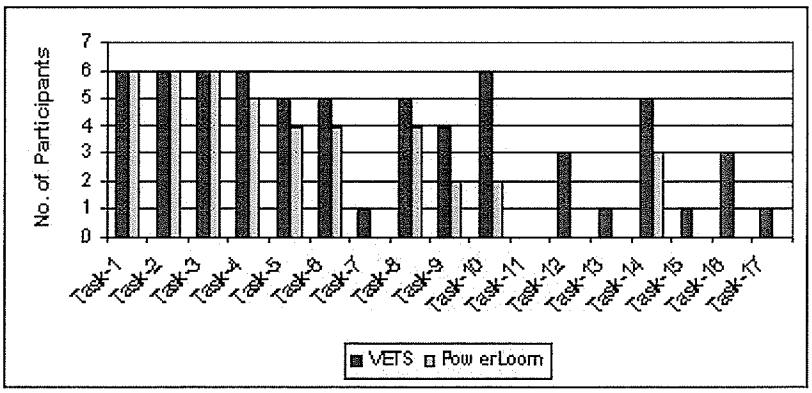

Figure 7.1: Task Completion Comparison Between Two Groups in Experiment 1

significantly different in terms of the number of correct answers. This value confirms the

hypothesis 1 that VETS users were able to perform more tasks correctly compared to the PowerLoom system users.

Number of Errors

Table 7.4 shows that in the PowerLoom system, participants made fewer errors compared to those using the VETS system. In PowerLoom system, the average number of errors for a user was 19, whereas in VETS, the average number of errors was 26. From the Table 7.2 we can see that there is no significant difference between two system users in terms of the number of errors. The negative t-value indicates that VETS users made more errors. As the difference is not significant we can state that we obtained this result by chance. Consequently, we cannot prove hypothesis 2.

Table 7.4: Number of Errors Users Made in Experiments 1

System Name	Number of Errors
VETS	153
PowerLoom	111

Number of Attempts

In VETS, participants made 244 attempts in total. On the other hand, in PowerLoom, participants made 181 attempts to solve tasks. In VETS, users' average number of attempts to solve all tasks was 41 compared to 30 of PowerLoom participants. In VETS, participants' average number of attempts was 1.33 times higher than that of PowerLoom system. The t-Test result from Table 7.2 shows that there is a significant difference between VETS and PowerLoom system users in terms of the number of attempts. The negative t-value indicates that VETS user made more attempts. This test result rejects hypothesis 3.

Time Taken

In VETS, all users together spent 225 minutes in total and in PowerLoom system users spent 197 minutes. Each participant spent on average 38 minutes in a VETS session. On the other hand, in PowerLoom each user spent 33 minutes on average. The negative t-value from Table 7.2 for time taken shows that VETS users took more time. However, this result was not statistically significant thus the scenario could have happened by chance. As a result, we cannot prove hypothesis 4.

Other Findings

Number of Tasks not Attempted

There is a significant difference between VETS users and PowerLoom users in terms of number of tasks not attempted. In VETS, we find 5 cases where participants did not attempt a task whereas in PowerLoom we found 21 such cases. In the Table 7.2 we can see that t-Test value gives a significant result for the tasks not attempted for two system users.

Number of Correct Answers Given for Different Task Categories

The number of correct answers in VETS and in the PowerLoom system for task category 1 was 34 and 31 respectively. From a t-Test (unrelated), t-value was 0.958315, which is not significant. On the other hand, for task category 2 (which are tasks with a medium difficulty level) VETS users had 20 correct answers, whereas PowerLoom users had 11 correct answers shown in Table 7.5.In this Table C refers to task category. The t-Test was used and the t-value was 2.576693, which is significant at $p < .025$ with a critical value 2.228 and 10 degrees of freedom. For task category 3 (task difficulty level 3) the performance of VETS and PowerLoom users also varied significantly. Even though for task category 1 there was no performance difference, for task category 2 and task category 3, VETS users performance was significantly better than that of PowerLoom users. Users' task success rate for the 3 categories of tasks are shown in Figure 7.2 for VETS and the PowerLoom system.

Table 7.5: Task Success Rate for Different Task Categories

Task Category	Total Tasks	Correct Answers on First Attempts	Correct Answers after Incorrect First Attempt	Tasks Attempted	Number of Correct Answer	Tasks Success Rate
C1 (VETS)	36	27	7	36	34	94%
C1 (PL)	36	25	6	36	31	86%
C2 (VETS)	24	9	11	24	20	83%
C2 (PL)	24	8	3	20	11	45%
C3 (VETS)	42	1	9	38	10	23%
C3 (PL)	42	0	0	25	0	0%

7.2 Experiment 2

For Experiment 2 a within subjects experiment design [12] is followed where same group of participants performed equivalent tasks in different conditions where their performance measure and compare for different dependent variables. In Experiment 2, to evaluate the effectiveness of VETS the same group of participants used both the systems VETS and PowerLoom for two sets of questions which were equivalent. In this experiment, half of the participants used PowerLoom in the first phase and VETS in the second phase and the other half people did the opposite. This experiment was conducted to eliminate individual differences between experimental conditions. The experimental setup, procedure and results are discussed below.

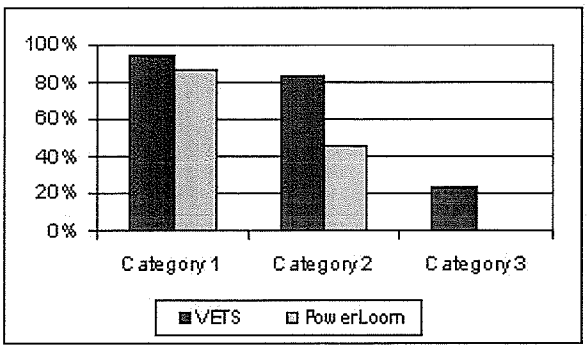

Figure 7.2: Task Success Rate for Different Task Categories

7.2.1 Hypothesis

On the basis of the ITS evaluations [29, 12] it is anticipated that

1. Participants in a VETS session will perform better compared to their session in the PowerLoom system in terms of the number of correct answers.

2. Participants will perform better in their VETS session independently of whether they used VETS in the first phase or in the second phase of their sessions.

3. Participants will take more attempts to perform tasks in their sessions with VETS compared to in their PowerLoom session.

7.2.2 Materials

The experiment was carried out at the University of Manitoba computer science graduate labs using the system VETS and the PowerLoom system.

7.2.3 Participants

Eleven graduate students and one under graduate student participated voluntarily in this test. Nine students were from the Computer Science department, two students from the

Electrical Engineering department, and one student from the Mathematics department. None of these students had prior knowledge in Description Logic.

7.2.4 Design

Experiment 2 was conducted using two systems, VETS and PowerLoom, to measure which system is more effective, where VETS had an advantage over the PowerLoom system because of VETS's verbalization. In this experiment, all users used both systems. Half the users used the PowerLoom system first and half of the users used the VETS system first to assure that none of the systems received advantages by being introduced second to the users. Two different question sets were used, which were equivalent with respect to task difficulty level.

7.2.5 Procedure

In Experiment 2, each student solved one set of tasks (see Appendix D) using VETS and the another set of tasks (see Appendix D) using the PowerLoom system. Both these task sets were comparable. Each set had 10 questions; tasks 1 to 5 were category 1 tasks (task difficulty level 1) and tasks 6 to 10 were category 2 tasks (task difficulty level 2). To ensure the equality of the two systems, half of the users used VETS in the first phase and the other half people used PowerLoom in the first phase. Moreover, to confirm the equality of two phases, 3 users used question set 1 in their first phase using VETS, and another 3 users used question set 2 in their first phase using VETS. This process was likewise followed for PowerLoom. Participants were randomly assigned to these phases.

An orientation to Description Logic was delivered for 10 minutes followed by a 5 minute demonstration of both systems. Then users practiced sample questions using both systems. After the practice session, users performed the real session. Users spent an average of 15 minutes on the PowerLoom system and 17 minutes on VETS. There was no time limit on any session. User performance data: the number of correct answers, the

number of attempts taken, the number of errors, and time taken to solve tasks, was stored in log files. For each student, two log files were created: one log file to record student performance using VETS and the other log file for performance using the PowerLoom system. After the evaluation, these files were analyzed to assess VETS's effectiveness.

7.2.6 Results

To measure the effectiveness of VETS, the experiment data was stored in tabular format and analyzed with respect to the following criteria.

Comparison of Number of Correct Answers for Both systems for Each Task Users can complete more tasks correctly using VETS compared to their sessions in PowerLoom as shown in Table 7.6. In VETS, each user corrected on average 6 tasks properly out of their 10 tasks. On the other hand, in PowerLoom users corected on average 4

Table 7.6: Number of Correct Answers in Experiment 2

System Name	Number of Correct	Tasks Success Rate
VETS	69	57%
PowerLoom	43	35%

tasks correctly. A comparison of the number of correct answers for each user for both systems is shown in Figure 7.3. The t-Test (related) shows that users performance varied significantly at $p < .005$ with a critical value 3.106 and degrees of freedom 11. This result satisfies hypothesis 1; VETS users perform better compared to the PowerLoom users in terms of the number of correct answers.

In this experiment, the number of correct answers in both phases of 6 participants who used PowerLoom in the first phase and VETS in the second phase were recorded

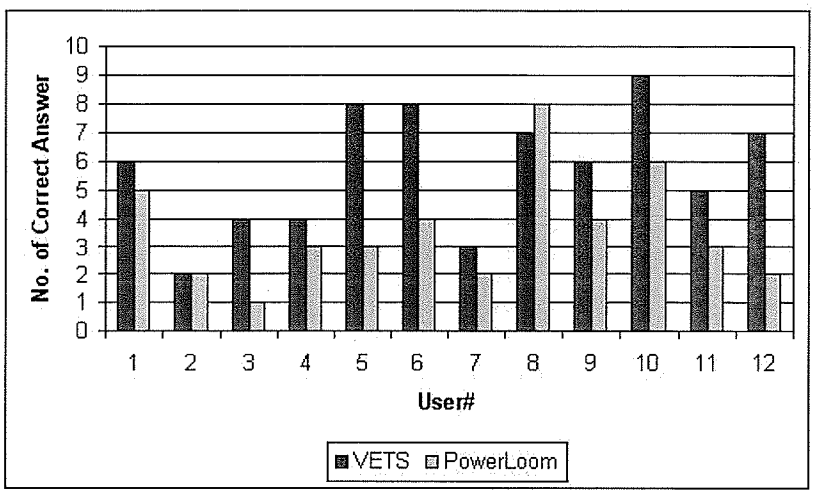

Figure 7.3: Task Success Comparison in Experiment 2

separately. The count values showed that users made more correct answers in their VETS session. The t-Test (related) shows that these two systems varied significantly at $p < .025$ with a critical value of 2.228 and 5 degrees of freedom. For this experiment, the number of correct answers given by participants who used VETS in the first phase and PowerLoom in the second phase were also recorded. The above t-Test gives the same result for this case also. The outcome of these two measurements proves that hypothesis 2 is true. We can conclude that users could complete more tasks correctly using VETS irrespective of the order in which they used the two systems.

Number of Attempts

The number of attempts users took to solve a task using the PowerLoom system and VETS was recorded in this experiment. In VETS, all users together took 254 attempts and in the PowerLoom sessions, users took 173 attempts in total. The t-Test (related) shows that these two systems varied significantly at $p < .025$ at 2.228 and 5 degrees of freedom. This result confirms hypothesis 3. VETS users made more attempts to solve tasks instead of giving up.

Other Findings

Correlations Between Number of Correct Answers and Number of Attempts

The Pearson Correlation test [12] was applied to test whether there was a correlation between the number of correct answers and the number of attempts. In this test, the R value shows that the above two variables are correlated at $p < .025$ with a critical value of .5760 and degrees of freedom 10. In VETS users were able to complete more tasks correctly probably because they made more attempts to solve tasks.

Correction Events

The data from Experiment 2 was also analyzed to check in how many cases with the

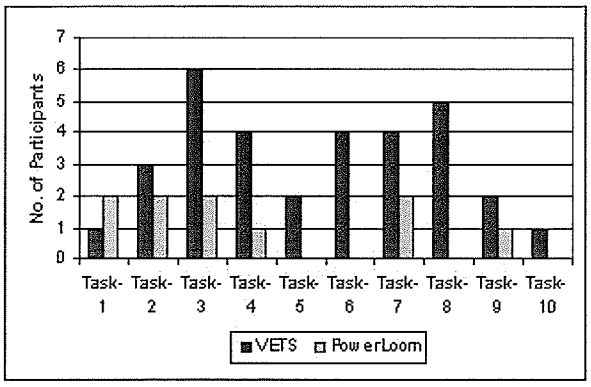

Figure 7.4: Error to Correct Answers in Experiment 2

help of the system's feedback, VETS user could correct their erroneous input. This value was also collected for PowerLoom users. For VETS, 32 such cases were found, and for PowerLoom, only 10 such cases were found. The t-Test(related) shows that these two systems varied significantly. The t-value showed significance at $p < .025$ with a critical value of 2.201 and degrees of freedom 11. Figure 7.4 shows the comparison between both systems indicating the number of participants for each task who were able to correct their errors. The results indicate that the VETS system helps users to correct their errors and complete task successfully.

Task Completion Rate

From this experimental data, the task completion rate [24] in both systems were calculated. The task completion rate helps to measure whether the system motivates users to work or discourage students, leading them to drop out. To measure task completion rate, how many cases users attempt all tasks in their session and how many users quit the session before trying all the tasks were recorded. In VETS, in 9 cases users attempted all tasks and 3 users did not. On the other hand, in PowerLoom, 5 users attempted all tasks and the remaining 7 users did not. The task completion rate in VETS was 75% and in PowerLoom was 41%, where the task completion rate as calculated using:

$$Task\ Completion\ Rate = \frac{number\ of\ users\ who\ completed\ their\ sessions}{total\ number\ of\ users}$$

As these data is nominal, one user can belong to only one of the two categories, either attempts all tasks or not. As a result, to measure the significance of these variables, a CHI-Square test was applied but the $\chi 2$ value shows that there is no significant difference between the two systems in terms of task completion rate. It could be due to small sample set we did not get any significant difference between the two systems in terms of task completion rate.

7.3 Experiment 3

The objective of this experiment was to measure the performance improvement over time using VETS as suggested by researchers [24]. In Experiment 3, the same group of participants used VETS in 2 sessions and their performance was measured in terms of the number of correct answers, the number of errors, the number of attempts and the time taken. The experiment set up, procedures, and results are discussed in this section.

7.3.1 Hypothesis

According to the results of the pre-test and post-test [29] of ITS, the following are anticipated:

1. If we provide tasks with the same difficulty level to the same group of users in 2 sessions users will perform better in the second session. They will be able to answer more tasks correctly.

2. If we provide tasks with the same difficulty level to the same group of users in 2 sessions users will require less time to solve tasks in the second session.

7.3.2 Material

Experiment 3 was conducted at one of the University of Manitoba computer science graduate lab. The VETS system was used to measure user performance improvement over time.

7.3.3 Participants

Six graduate students from the Computer Science department voluntarily participated in this test. None of these users had prior knowledge in Description Logic languages before their first session.

7.3.4 Design

A multi-session experiment design was followed to measure user performance improvement over time. Each user attended two sessions within a period of one week in which they participated in two comparable tests on Description Logic.

7.3.5 Procedure

Six students participated in two sessions within a one-week period where they performed some equivalent tasks. In both sessions, the number of tasks and difficulty level of the tasks were similar. Before the first session, a short introduction of Description Logic was given for 10 minutes followed by the 5 minute system demonstration. After the demonstration, users participated in the real session; the duration of the session was on average 17 minutes. For the second session, users participated in the real test without any introduction or demonstration. The duration of this session was on average 13 minutes. There was no time restriction for the experiment. In this experiment, the number of correct answers and the time taken to solve a problem were stored in separate log files for each user. There were ten questions in each session. Questions for session one is given in Appendix D (set 1) and session two in Appendix E.

7.3.6 Results

All information for the 6 users was tabulated and compared for each user's performance in the two sessions. The tabular form is shown in Table 7.7.

Table 7.7: Experiment 3 Observation

	User1	**User2**	**User3**	**User4**	**User5**	**User6**
Session 1						
No. of Correct Answers	4	5	6	3	4	7
Time Taken	16.85	19.13	20.22	15.47	13.72	14.23
Session 2						
No. of Correct Answers	7	8	9	7	7	9
Time Taken	13.57	16.08	17.38	9.40	9.63	13.25

Number of Correct Answers

Users performed more tasks correctly in the second session compared to the first session. Table 7.7 shows that all 6 students performed better in the second session. The t-Test (related) shows that user performance varied significantly at $p < .0005$ with a critical value of 6.859 and the degrees of freedom 5. As a result, hypothesis 1 holds. In Figure 7.5 user performance in 2 sessions in terms of the number of correct answers are shown.

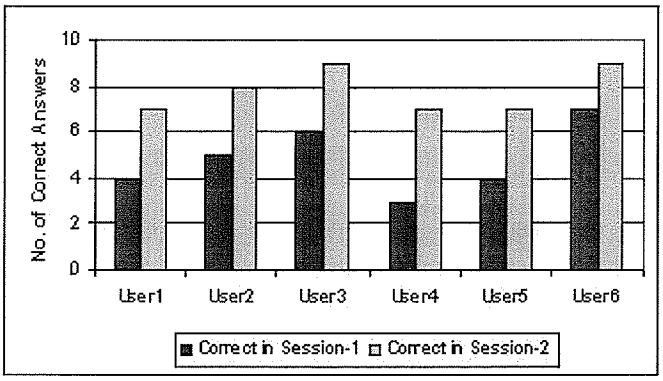

Figure 7.5: Number of Correct Answers Over Two Sessions

Time Taken

Users took less time to answers tasks in the second session compared to the first session. Table 7.7 shows that all 6 students performed better in the second session in terms of the time taken. The t-Test (related) shows that user performance varied significantly at $p < .005$ with a critical value of 4.032 and the degrees of freedom 5. This result confirms hypothesis 2.

7.4 User Satisfaction Measurement

Each participant who used VETS was administered a post-experiment questionnaire (see Appendix F). The questionnaire consisted of one Likert-scale and seven open-ended [26]

questions. Thirteen users out of twenty four participated in this process. All user comments were stored anonymously. The analysis of the questionnaire is given below:

Usability and Error Feedback

We tried to evaluate VETS' usability as a tutoring tool based on users' open ended comments. Some positive comments were: "It is very user friendly;" "Pretty neat interface and easy to use;" "Nice interface (look and feel);" "Nothing is missing. I think the system is fine." Some participants provided suggestions and other offered criticism such as that they did not like the "start button" in the interface. However, this button was included only for the evaluation to keep track of the current task users were performing. Some of the suggestions were: "provide syntax"; "give examples"; " categorize users and provide help accordingly".

We also tried to evaluate how helpful the error feedback was. On error feedback, there were two questions (3 and 4) in the questionnaire. Out of 26 comments 19 were positive, two comments were "reasonably good", one participant was uncertain, three found the error feedback "not very helpful" and one commented " I did not depend on error feedback".

The Helpfulness of the Verbalization

To evaluate how helpful the verbalization was, both Likert-scale and open-ended questions were used. For the Likert-scale question there were five categories for opinions: strongly agree to strongly disagree. The Likert-scale shows that 30% of users strongly agreed, 62% of users agreed and 8% of users were neutral on the statement that "*Verbalization is helpful to learn the Description Logic language PowerLoom*". The results are shown in Figure 7.6.

Open-ended comments by the participants also suggested that verbalization is helpful to learn PowerLoom. In the questionnaire, there was one open-ended question (question number 1) on verbalization. In this question we received ten positive feedbacks and three

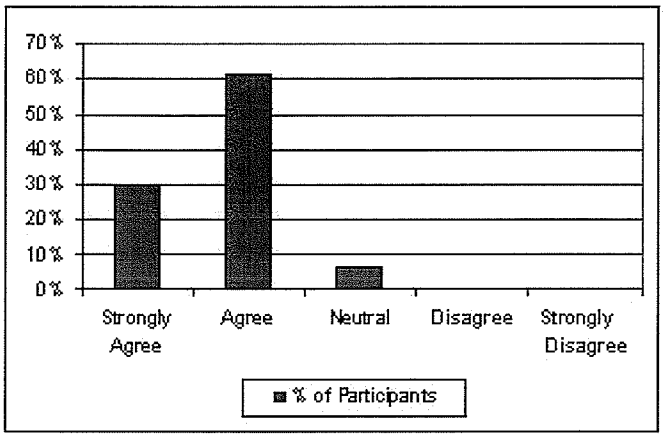

Figure 7.6: Users' Response Regarding the Usefulness of Verbalization

neutral feedback. Some positive comments were:
"Verbalization helps to understand mistakes"; "I liked the way it replied to my input"; "Verbalization is excellent and very much helpful for the users"; "The verbalization part is very good. It is helpful for a user". Some neutral comments were:
" I guess so"; "A little bit".

Other Comments

Question 5 to 8 were regarding users' satisfaction measurement about the system and their suggestions for future improvement of VETS.

In response to question 5 "What do you like about the system and what you dislike and why?" we received 8 positive comments, one neutral comments, two suggestion, and three criticisms. 2 positive feedback on verbalization were it helped them to learn PowerLoom. 6 positive feedback were on user interface of VETS. One user commented that the system was ok. One suggestion was VETS should improve the keyboard interaction and the other suggestions was error feedback needs to be more interactive. One criticism was that in VETS users need to perform more operations to finish a task than in PowerLoom. Another criticism was that users need to press start button to start a new task. However, this button was included in VETS only for evaluation purpose. Another criticism was

that PowerLoom syntax is complicated.

For question 6 "Which part of the system should be improved?" we received 9 suggestions, 3 criticisms. One user did not answer this question. Five of these suggestions was regarding error feedback improvement. One user suggested that VETS should have command guide (i.e. readme or help file). Another user suggested that VETS should include extra initial instructions to the beginners. Another suggestion was the test design should be improved. Another suggestion was that in VETS keyboard interaction should be improved. All three criticisms regarding the user interface such as that VETS is not user friendly enough.

For question 7 "What do you think is missing in the system?" we got 6 suggestions, one criticism. 4 users were satisfied with the system; according to them nothing is missing in VETS. Two users did not answer this question. One user suggested to add timing per question; another user suggested whether the system can generate syntax for users; 2 users suggested to improve feedback; one user suggested to test VETS in huge amount; one suggested to summarize user input and output. The criticism was regarding the user interface that user needs to enter the current task number.

In response to question 8 "Do you have any comments about the system?" we received 2 positive comments, three suggestions. 6 users answers were "no comment" and 2 users did not answer. One positive response was that VETS is user friendly and the other was good addition to PowerLoom. One user suggested that VETS needs to improve its user interface; one commented that input and output should be in different color; one suggested to categorize users and provide help according to the user category.

7.5 Discussion

Three quantitative experiments and a qualitative evaluation through a questionnaire have been used to evaluate the effectiveness of VETS. In Experiment 1 and Experiment 2, participants who used VETS were able to solve more tasks correctly compared to

the participants who used PowerLoom system. This outcome was consistent no matter whether they used VETS in the first phase or in the second phase of their sessions period.

Experimental results also show that there is no significant difference between two systems in terms of the number of errors user made and the time taken by the users but there is a significant difference between two system in terms of the number of correct answers. Experimental result also show that VETS users made more attempts to solved tasks correctly. From the statistical test we also found that the number of attempts and the number of correct answers are correlated. We can interpret this outcome that without any significance difference in terms of the number of error and the time taken VETS users performed better compared to the PowerLoom users because they made more attempts which lead more correct answers. The reasons behind VETS users making more attempts could be *a)* VETS increases user persistence or motivation by providing feedback as some form of encouragement which helped them make more attempts and solve tasks correctly or *b)* VETS users understood better what they are doing (we received some positive feedback regrading this from the open-ended questions from the questionnaire) that helped them to be persistent and achieve more correct answers. On the other hand, PowerLoom users might have difficulties to understand PowerLoom formulas due to the lack of proper feedback which discourage them to continue and finish given tasks.

Experimental results regarding the difficulty levels of tasks show that for task category 1, which has level 1 task difficulty, VETS and PowerLoom users performed equally, which means there was no significant difference between two systems. But for task category 2 and 3, user performance varies significantly. VETS users performed significantly better in task categories 2 and 3 compared to PowerLoom users. We can conclude that people do not need VETS tutoring for task category 1, but they need help for task category 2 and 3.

In Experiment 3, we found that for the same difficulty level tasks users were able to make more tasks correctly in their second session. In the second session, users also took less time compared to their first session. This result could be due to performance

improvement over time enabling them to recall and use their learned information and skills better.

In qualitative experiment, we used Likert-scale and open-ended questions to determine user satisfaction. Users' responses to Likert-scale questions show that participant agreed that verbalization helped them to learn PowerLoom language. Responses from the open-ended questions also showed that VETS achieved user satisfaction.

Chapter 8

Discussion of the Thesis

8.1 Complexity of Work

Challenges of tasks are listed below:

- I needed to ascertain a suitable way for verbalizing Description Logic / PowerLoom formulas using appropriate natural language generation techniques. One main aspect of this was finding a proper way for content selection and sentence generation. I also needed to consider syntactical issues such as subject-verb agreement, determinant selection, and proper use of pronouns.

- For sentence generation as part of the verbalization process, I needed to design and implement a unification grammar. Currently available shareware implementations of unification grammars are incompatible with the PowerLoom system.

- I needed to re-design and extend the PowerLoom grammar because the available PowerLoom syntax description is incomplete. I contacted the PowerLoom developers but they were unable to provide a complete grammar. The PowerLoom syntax serves as a basis for analyzing user input and also for generating complex verbalizations, in accordance with the syntactic structure of a PowerLoom construct.

- I needed to develop a tutoring system to show the effectiveness of the verbalization in the context of tutoring systems. Typical tutoring systems consist of some essential components such as a user model, domain module, and instruction module. I needed to design and build a tutoring system from scratch because I did not find any suitable shell system to develop my tutoring system.

- Another challenge of my work was to evaluate my system properly. I needed to design a study with human participants, for example, fellow students. This involves the design of appropriate tasks and questionnaires, as well as the definition and used of suitable evaluation parameters (such as the number of errors per task, the time spent on tasks, the number of completed tasks). This study allowed to compare users performance who used VETS with the PowerLoom system. Furthermore, I needed to schedule and arrange sessions with human participants for this system evaluation which was time consuming and exhaustive.

8.2 Limitations of the System

1. Even though it is possible to verbalize both the simple and complex DL formulas, if the formula becomes really large, there may be some repetitions of similar phrases in output verbalized text output, which may result in ambiguity. My system generates single sentence output for a DL formula. In reality, if we have an extensive DL formula, we need to divide the information in the formula and generate two to three sentences, using links among those sentences. However, I did not consider doing this. Usually people do not enter long (or complicated) DL formulas because this is also cumbersome for them. Moreover, users perform the same task by defining two or three simple formulas instead of defining a complicated large formula.

2. Users must define feature names starting with "has" for example, "has_size". "has_name" instead of defining some meaningless name such as "blub" to make

the verbalized output meaningful.

8.3 Implementation Issues

- In DL formula, at one point different errors can occur due to violating different PowerLoom grammar rules. For example: for this formula (*defconcept chair ?x furniture*)) we can see that an opening parenthesis is missing after chair. Once the system parses this formula it will detect that there is a syntax error. But at this point different possible constructs can appear in different rules. Therefore, to identify the error that an opening parenthesis is missing, the system needs to parse the remaining part of the formula *?x furniture*)). To handle user errors properly, I needed to design the algorithm properly.

- To generate coherent and semantically correct verbalization output, I need to design subject-verb agreement algorithm. To use the relative clause and referential expression properly I needed to design the unification grammar accordingly.

8.4 Other Issues

No Suitable Shell Available

It would have been possible to use a shell to develop the tutoring system and incorporate the verbalization component into this if there existed a suitable shell, which would save the implementation time. I searched for a shell system but did not find any suitable shell. For example, a shell called "CTAT" [2] is available to develop ITS, but I found this shell was not suitable to develop my system because this shell supports only template-based text generation. As a result, I needed to develop all the components of VETS from scratch.

Grammar-based Generation vs. Template-based Generation

VETS used unification grammar instead of templates because the number of templates would increase potentially with the different forms of DL formulas because for each DL formula a corresponding template needs to be defined. Using NLG techniques with a unification grammar, DL formulas can be verbalized using a small set of rules such as rules for noun phrase, verb phrase. Verbalization texts which are generated by unification grammar can provide additional relevant information but, in template-based text this is not possible.

Chapter 9

Conclusion

Tutoring systems help people in their learning by providing error feedback and suggestions. However, these systems have yet to achieve complete user satisfaction. Researchers are trying to improve the effectiveness of tutoring systems by incorporating various techniques such as natural language dialogue, and human computer adaptive learning environment. There are some tutoring systems available for logic such as Propositional Calculus and First Order Predicate Logic. Currently available logic tutoring systems provide tasks, feedback, and suggestions in order to guide users, but none of them provides help to understand the meaning of logic formulas in natural language. Usually users have difficulties in understanding logic formulas but they can easily understand corresponding natural language text. A tutoring system that can communicate with users, accompanied by logic constructs and natural language interpretation of each of those constructs, would be helpful. Verbalization of formal structures has already been successfully applied in computer generated proofs but has not yet been explored in the field of tutoring systems.

I incorporate a new feature for Intelligent Tutoring Systems: verbalization of user input for ITS with formal, logic-like languages as application domains. My system, the Verbalization Enhanced Tutoring System VETS, verbalizes logic constructs and thus

helps users to better understand logic formulas. The generated text provides additional feedback to articulate users what they are doing. By using verbalization techniques, VETS can also describe knowledge base information as text, to guide users to build the knowledge base. VETS uses Natural Language Generation (NLG) techniques for verbalization and error feedback.

I have evaluated the effectiveness of VETS using quantitative studies and have measured user satisfaction using questionnaires. To evaluate VETS quantitatively, I used the system VETS and stand alone PowerLoom system using the same sets of tasks. Evaluation results from quantitative studies show that there is no significant difference between two system users in terms of the number of error users made and the time taken. VETS users performed significantly better compared to PowerLoom users in terms of the number of correct answers. Experimental results also show that VETS users made more attempts to complete a task correctly instead of giving up, which at the end enabled them performed more tasks correctly. VETS user task success rates are also high, and the difference of the task success rate between the two system users increases when the task difficulty level increases. VETS user can correct more of their erroneous input using system's help compared to PowerLoom users. Experimental results also show that users performance improved over multiple sessions in terms of the number of correct answers and the time taken. Structured questionnaires showed that VETS achieved greater user satisfaction.

Future Work

In future I want to

- maintain an ontology for different classes of objects where a definition of each object class will be stored. For example, the ontology for chair will store all the basic features of a chair, and it will also store information how the singular plural varies for this concept. Other relevant information of this class will also be stored

in the ontology. As a result, the verbalization output will be able to include the system knowledge about this particular object;

- enable VETS to generate different types of verbalization output for a particular DL formula. For example the formula:
  (*defconcept chair*(?*x furniture*))
  We can generate verbalization using a meta language :
  You defined a concept named "chair" which is a sub concept of another concept called furniture.
  We can also generate verbalized output using more human-like language:
  Chair is a furniture.
  I also want to measure what kinds of verbalized output helps users to perform better.

Bibliography

[1] D. Abraham, L. Crawford, L. Lesta, A. Merceron, and K. Yacef. The logic tutor: A multimedia presentation. *Interactive Multimedia Electronic Journal on Computer Enhanced Learning*, 3(2), 2001.

[2] V. O. Aleven, O. Popescu, and K. R. Koedinger. Pedagogical content knowledge in a tutorial dialogue system to support self-explanation. In *Proc. Workshop on Tutorial Dialogue Systems in AIED'01*, pages 59–70, 2001.

[3] M. Alexoudi, C. Zinn, and A. Bundy. English summaries of mathematical proofs. In *Proc. Workshop on "Computer-supported mathematical theory development" at the International Joint Conference on Automated Reasoning (IJCAR)*, pages 49–60, 2004.

[4] L.A. Becker and X. Huang. An intelligent tutor for normal form determination. In *Proc. The 19th SIGCSE technical symposium on Computer science*, pages 205–209, 1998.

[5] P. Brusilovsky, E. Schwarz, and G. Weber. Elm-art: An intelligent tutoring system on world wide web. In *Proc. ITS'96*, pages 58–64, 1996.

[6] B. Cheung, L. Hui, J. Zhang, and S. M. Yiu. Smarttutor: an intelligent tutoring system in web-based adult education. *Journal of Systems and Software*, 68(1):11–25, 2003.

[7] C. Desmoulins and N.V. Labeke. Towards student modelling in geometry with inductive logic programming. In *Proc. EuroAIED'96*, pages 94–100, 1996.

[8] B. D. Eugenio and M. J. Trolio. Can simple natural language generation improve intelligent tutoring systems? In *Proc. Fall Symposium on Building Dialogue Systems for Tutorial Applications in AAAI'00*, pages 96–104, 2000.

[9] D. Fensel, F. V. Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Pater-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[10] R. Freedman, S. S. Ali, and S. McRoy. Links: What is an intelligent tutoring system? *Intelligence Archive*, 11(3):15–16, 2000.

[11] E. Goldberg, N. Driedger, and R. I. Kittredge. Using natural-language processing to produce weather forecasts. *In Communications of the ACM*, 38(11):71–79, 1995.

[12] J. Greene and M. D'Oliveira. *Learning to use statistical tests in psychology: a student's guide.* Open University Press, 1982.

[13] H. A. Güvenir. An object-oriented tutoring system for teaching sets. *ACM SIGCSE Bulletin archive*, 27(3):39–46, 1995.

[14] S. H. Hahn, J. S. Song, K. Y. Tak, and J. H. Kim. An intelligent tutoring system for introductory c language course. *Computer & Education*, 28(2):93–102, 1997.

[15] A. M. Holland-Minkley, R. Barzilay, and R. L. Constable. Verbalization of high level formal proofs. In *Proc. American Association for Artificial Intelligence*, pages 277–284, 1999.

[16] PowerLoom Homepage. http://www.isi.edu/isd/LOOM/PowerLoom/.

[17] X. Huang and A. Fiedler. Proof verbalization as an application of nlg. In *Proc. the 15th International Joint Conference on Artificial Intelligence*, pages 965–970, 1997.

[18] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall, 2000.

[19] C. Kemke and S. Mithun. Development of an intelligent tutor for description logics. In *Proc. Workshop on Applications of Description Logics in KI'04*, 2004.

[20] J. H. Kim. *Natural Language Analysis and Generation for Tutorial Dialogue.* PhD thesis, Illinois Institute of Technology, 2000.

[21] Kinshuk, A. Patel, and D. Russell. A multi-institutional evaluation of intelligent tutoring tools in numeric disciplines. *Educational Technology & Society*, 3(4), 2000.

[22] G. Knight, D. Kilis, and P. C. Cheng. An architecture for an integrated active help system. In *Proc. ACM Symposium on Applied Computing*, pages 58–64, 1997.

[23] S. Lukins, A. Levicki, and J. Burg. A tutorial program for propositional logic with human/computer interactive learning. In *Proc. 33rd SIGCSE technical symposium on Computer science education*, pages 381–385, 2002.

[24] A. M. Mark and J. E. Greer. Evaluation methodologies for intelligent tutoring systems. *Journal of Artificial Intelligence in Education*, 4(2):129–153, 1993.

[25] D. Nardi and R. J. Brachman. *The Description Logics Handbook*, chapter An Introduction to Description Logics, pages 1–40. Cambridge University Press, 2003.

[26] J. H. Obradovich, P. J. Smith, S. A. Guerlain, S. Rudman, and J. W. Smith. Field evaluation of an intelligent tutoring system for teaching problem-solving skills in transfusion medicine. In *Proc. IEA/HFES'00*, pages 338–341, 2000.

[27] F. Peinado, P. Gervás, and B. Díaz-Agudo. A description logic ontology for fairy tale generation. In *Proc. Workshop on Language Resources for Linguistic Creativity*, pages 56–61, 2004.

[28] N. K. Person, A. C. Graesser, R. J. Kreuz, V. Pomeroy, and The Tutoring Research Group. Simulating human tutor dialog moves in autotutor. *International Journal of Artificial Intelligence in Education*, 12:23–39, 2001.

[29] L. M. Razzaq and Heffernan. Tutorial dialog in an equation solving intelligent tutoring system. In *Proc. ITS'04*, pages 851–853, 2004.

[30] N.C. Rowe. A computer tutor for logic semantics. In *Proc. Frontiers in Education Conference (FIE)*, 1999.

[31] O. C. Santos, A. Rodríguez, E. Gaudiosoa, and J. G. Boticario. Helping the tutor to manage a collaborative task in a web-based learning environment. In *Proc. Workshop on Towards Intelligent Learning Management Systems in AIED'03*, 2003.

[32] A. Simón, A. Martínez, M. López, J. A. Maestro, J. M. Marqus, and C. Alonso. Learning computational logic with an intelligent tutoring system: SIAL. In *Proc. 1st International Congress on Tools for Teaching Logic*, 2000.

[33] E. R. Sykes and F. Franek. A prototype for an intelligent tutoring system for students learning to program in Java TM. In *Proc. IASTED International Conference on Computers and Advanced Technology in Education*, pages 78–83, 2003.

[34] B. Tekinerdoğan. Design of a reflective tutoring system shell. Technical report, Dept. of Computer Science, University of Twente, The Netherlands, 1995.

[35] Y. Zhou and M. W. Evens. A practical student model in an intelligent tutoring system. In *Proc. 11th International Conference on Tools with Artificial Intelligence*, pages 13–18, 1999.

Appendix A

Sample Verbalization Text

1. (*defconcept furniture*)
   Verbalization: Furniture is a concept.

2. (*defconcept movable_furniture*(?*f furniture*))
   Verbalization: Movable_furniture is a furniture.

3. (*assert*(*furniture chair*1))
   Verbalization: Chair1 is a furniture.

4. (*defconcept sizes*(?*s*) :<=> (*member* − *of* ?*s*(*setof small big*)))
   Verbalization: Sizes is a concept which has values small and big.

5. (*defrelationhas_size*((?*f furniture*)(?*s sizes*)))
   Verbalization: Has_size is a relation between furniture and sizes.

6. (*assert*(*has_size chair*1 *small*))
   Verbalization: Chair1 has_size small.

7. (*ask*(*has_size chair*1 *small*))
   Verbalization: Ask whether the chair1 has_size small.
   Query Output: TRUE

8. (*retrieve all* ?*x*(*furniture* ?*x*))
   Verbalization: What are the instances of concept furniture?
   Query Output: There is 1 solution:♯1: $?X = CHAIR1$

9. (*defrelation has_name*((?*c company*)(?*n string*)))
   Verbalization: Company has_name which will be a string.

10. (*assert*(*has_name Acme*"*Acme_Cleaners*"))
    Verbalization: Acme has_name "Acme_Cleaners".

11. (*deffunction has_employees*((?*c company*)) : − > (?*i integer*))
    Verbalization: Has_employees of a company must be an integer value.

12. (*ask*(= (*has_employees Acme*)10))
    Verbalization: Ask whether has_employees of Acme is equal to 10.
    Query Output: TRUE

13. (*defconcept corporation*((?*c company*)) :<=> (*and*(*company* ?*c*)(< (*has_employees* ?*c*)200)))
    Verbalization: Corporation is a company if and only if it is a company and it has_employees less than 200.

14. (*assert*(= (*has_employees Megasoft*)1000))
    Verbalization: Megasoft has_employees equal to 1000.

15. (*retrieve all* ?*x*(*and*(*company* ?*x*)(< (*has_employees* ?*x*)50)))
    Verbalization: Retrieve variable x where x is a company and it has_employees less than 50.
    Query Output: There is 1 solution:♯1: $?X = ACME$

16. (*retract*(= (*has_employees Acme*)10))
    Verbalization: Delete that Acme has_employees equal to 10.

Appendix B

Sample Error Feedback

1. *defconcept furniture*)
   Error Occurred while compiling the DL construct. Generating error information...
   [At this position] current value = 'defconcept', a (is expected before this position.

2. (*fdefconcept furniture*)
   Error Occurred while compiling the DL construct. Generating error information...
   [At this position] current value = 'fdefconcept', a defconcept is expected.

3. (*defconcept furniture*
   Error Occurred while compiling the DL construct. Generating error information...
   (defconcept furniture [At this position] current value = 'null', a premature end to DL construct.

4. (*defconcept movable_furniture*(?*f table*))
   Error Occurred while compiling the DL construct. Generating error information...
   (*defconcept movable_furniture*(?*f* [At this position] current value = 'table', the referred concept name does not exist.

5. (*defconcept movable_furniture*(*furniture*))
   Error Occurred while compiling the DL construct. Generating error information...
   (*defconcept movable_furniture*([At this position] current value = 'furniture', a variable name is expected (for example ?x)

6. (*defconcept movable_furniture ?x furniture*))
   Error Occurred while compiling the DL construct. Generating error information...

(*defconcept movable_furniture ?x* [At this position] current value = 'furniture', a) is expected before this position.

7. (*defconcept sizes*(*?s*) :<=> (*member ?s*(*setof small big*)))
   Error Occurred while compiling the DL construct. Generating error information...
   (*defconcept sizes*(*?s*) :<=> ([At this position] current value = 'member', a member-of is expected.

8. (*deffunctionhas_employee*((*?c company*))(*?i integer*))
   Error Occurred while compiling the DL construct. Generating error information...
   (*deffunctionhas_employee*((*?c company*))[At this position] current value = '(', a :<=> is expected before this or , a :=> is expected before this or , a : is expected before this or , a : − > is expected before this or , a => is expected before this position.

9. (*deffunction has_employee*((*?c company*)) : − > (*?i*))
   Error Occurred while compiling the DL construct. Generating error information...
   (*deffunction has_employee*((*?c company*)) : − > (*?i*[At this position] current value = ')', an integer value is expected or , a string value is expected or , the referred concept name does not exist.

10. (*defconcept corporation*1((*?c company*)) :<=> (*and*(*company ?c*)((*has_employees ?c*)200)))
    Error Occurred while compiling the DL construct. Generating error information...
    (*defconcept corporation*1((*?c company*)) :<=> (*and*[At this position] current value = 'company', operator and is expected.

Appendix C

Questions for Experiment 1

Test Questions

VETS Evaluation

You will develop a knowledge base to represent a small "Business" model domain. You define concepts, relations, and functions to model the domain on the conceptual level. Then, you create instances of these concepts, relations, and functions to reflect concrete companies, their attributes, and relations. You modify and also query the knowledge base.
Please follow the sequence of tasks.

TASK♯1
Define a concept **company**

TASK♯2
Define another concept called **worker**

TASK♯3
Create an instance of the concept company called **ACME**

TASK♯4
Create an instance of the concept worker called **Mary**

TASK♯5
Define a relation **works_for** between concepts worker and company

TASK♯6
Assert to the knowledge base that Mary works_for the company ACME

TASK♯7
Retrieve the values for the relation works_for. (use the operator retrieve all)

TASK♯8
Define a binary relation called **has_name**, whose first argument must be an instance of the concept company and the second argument must be a string. (string is predefined)

TASK♯9
Assert that ACME has_name "**ACME_Cleaners**". (Hints: insert the name ACME_Cleaners as a string)

TASK♯10
Define a function named **has_employees**. This function has 2 arguments: the first one representing the company (instance of company) and the second one is an integer representing the number of employees of the company. (Hints: for defining function you need $:->$ operator between two arguments. Integer is predefined.)

TASK♯11
Enter into the knowledge base that ACME has_employees equal to 10. (Hints: use the operator "="; PowerLoom uses prefix notation.)

TASK♯12
Ask the knowledge base whether Acme has_employees equal to 10.

TASK♯13
Define **Corporation** as a sub concept of company, with has_employees less than 500.

Here, you have to define that corporation is a sub concept of company if and only if (using operator :<=>) it is a company and it has_employees less than 500. (Hints: use the operator "<"; PowerLoom uses prefix notation.)

TASK♯14
Create an instance of corporation called **Megasoft**)

TASK♯15
Assert that Megasoft has_employees equal to 400. (Hints: use operator "=")

TASK♯16
Retrieve all companies, which has_employees less than 20. (Hints: use operator "<")

TASK♯17
Delete the information that ACME has_employees equal to 10 using the retract construct.

Appendix D

Questions for Experiment 2

Set 1
Test Questions

VETS Evaluation

You will develop a knowledge base to represent human relation "likes" and human attribute "has_age". You define concepts, relations, and functions to model this domain on the conceptual level. Then, you create instances of these concepts, relations, and functions to reflect concrete human relation and attributes. You modify and also query the knowledge base.
Please follow the sequence of tasks.

TASK♯1
Define a concept **human**

TASK♯2
Define another concept **person** which is a sub concept of human.

TASK♯3
Define a relation **likes** between two persons.

TASK♯4
Create an instance of the concept person, named **Fred**

TASK♯5
Create another instance of the concept person, named **Joe**

TASK♯6
Assert to the knowledge base that Fred likes Joe

TASK♯7
Retrieve all instances of the relation likes. (use the operator retrieve all)

TASK♯8
Now, define a function has_age, whose first argument must be an instance of the concept person and the second argument must be an integer. (integer is predefined)

TASK♯9
Assign to the KB that Fred has_age 32.

TASK♯10
Retrieve all persons name who has_age less than 35 years (use operator "<").

Set 2
Test Questions

VETS Evaluation

You will develop a knowledge base to represent a small "Business" model domain. You DEFINE concepts, relations, and functions to model the domain on the conceptual level. Then, you CREATE instances of these concepts, relations, and functions to reflect concrete companies, their attributes, and relations. You modify and also query the knowledge base.
Please follow the sequence of tasks. If you leave out a task, mark it as not done.

TASK♯1
Define a concept **company**

TASK♯2
Define another concept called **worker**

TASK♯3
Create an instance of the concept company called **ACME**

TASK♯4
Create an instance of the concept worker called **Mary**

TASK♯5
Define a relation **works_for** between the concepts worker and company

TASK♯6
Assert to the knowledge base that Mary works_for the company ACME

TASK♯7
Retrieve the values for the relation works_for. (use the operator retrieve all)

TASK♯8
Define a binary relation called **has_name**, whose first argument must be an instance of

the concept company and the second argument must be a string. (string is predefined)

TASK♯9
Assert that ACME has_name "**ACME_Cleaners**". (insert the name ACME_Cleaners as a string)

TASK♯10
Ask whether ACME has_name ACME_Cleaners

Appendix E

Questions for Experiment 3

Test Questions

VETS Evaluation

You will develop a knowledge base to represent human relation "has_child" and "has_name". You define concepts, relations, and functions to model the domain on the conceptual level. Then, you create instances of these concepts, relations, and functions to reflect concrete human relation and attributes. You modify and also query the knowledge base.
Please follow the sequence of tasks.

TASK♯1
Define a concept **human**

TASK♯2
Define a concept **parent** which is a sub concept of human.

TASK♯3
Define another concept **child** which is also a sub concept of human.

TASK♯4
Create an instance of parent called **John**

TASK♯5

Create an instance of child called **Joe**

TASK♯6
Define a relation **has_child** between the concept parent and child

TASK♯7
Assign that John has_child Joe

TASK♯8
Now, define a relation **has_name**, whose first argument must be an instance of the concept human and the second argument must be a string. (string is predefined)

TASK♯9
Assign that John has_name "John Phillph"

TASK♯10
Retrieve all instances of the relation has_child. (use the operator retrieve all)

Appendix F

Questionnaire

DL Tutor Feedback
Since I am currently evaluating system Verbalized Enhanced Tutoring System (VETS), I would appreciate to get some feedback from those of you, who have used the tutor. Your answers will be kept confidential and in no form associated with you. The purpose of these questions is solely to evaluate VETS.

1. How do you find the verbalization part (translation of DL formula into English text)? Do you think it is helpful for a user when learning PowerLoom?
2. Verbalization is helpful to learn the Description Logic language PowerLoom
   a.Strongly Agree b.Agree c.Neutral d.Disagree e.Strongly Disagree
3. How good or helpful is the feedback?
4. Do you think the error detection and feedback are helpful in the knowledge base development?
5. What do you like about the system and what do you dislike and why?
6. Which parts of the system should be improved?
7. What do you think is missing in the system?
8. Do you have any other comments about the system?

Appendix G

Sample Log File

USERKK ♯error = 0 ♯correct = 1 ♯attempt =1
Time taken for task♯1 is 13.189 sec

USERKK ♯error = 1 ♯correct = 1 ♯attempt =2
Time taken for task♯2 is 32.625 sec

USERKK ♯error = 0 ♯correct = 1 ♯attempt =1
Time taken for task♯3 is 38.856 sec

USERKK ♯error = 2 ♯correct = 1 ♯attempt =3
Time taken for task♯ 4 is 73.189 sec

USERKK ♯error = 0 ♯correct = 1 ♯attempt =1
Time taken for task♯ 6 is 59.063 sec

USERKK ♯error = 1 ♯correct = 0 ♯attempt =1
Time taken for task♯7 is 85.401 sec

USERKK ♯error = 4 ♯correct = 0 ♯attempt =5
Time taken for task♯8 is 113.189 sec

USERKK ♯error = 2 ♯correct = 1 ♯attempt =3
Time taken for task♯9 is 131.189 sec