DEKADE II

An Environment for Development and Demonstration in Natural Language Processing

Jesse English
jesse.english@umbc.edu

0. Abstract

This paper presents the work done on the DEKADE II system. This system is an interface for developing and demonstrating the NLP analyzer OntoSem at the ILIT lab at UMBC. The system was designed from the ground-up to support browsing and editing of the static knowledge resources, as well as stepped usage of the analyzer for debugging and production of golden TMRs. DEKADE II was created to be a cross-platform client-server system, with thought given to data security, data synchronization, user experience and modular flexibility.

1. Introduction

The purpose of the DEKADE II (DII) project has been to design and implement a development, evaluation, knowledge acquisition and demonstration environment (hence DEKADE) for a natural language processing (NLP) system, OntoSem. OntoSem includes a variety of knowledge bases (static knowledge resources) and processors, which blend together to facilitate text analysis into an ontological-semantic representation of its meaning. The DII system must allow the user to monitor the execution of the various processors in OntoSem (with the goal of system tuning and debugging), run different applications of OntoSem as well as evaluation exercises and demonstrations, and, last but not least, efficiently support and facilitate the most labor-intensive kind of operation in OntoSem - the ongoing acquisition and maintenance of all of the static knowledge resources in the system. It is clear that for such a complex environment to be successful, each of the key components of the underlying system (OntoSem) must be easily accessible from one standardized location. However, before DII could be brought forth as a visual interface, it would need a strong supporting architecture, the DekadeAPI.

The DekadeAPI is an application programmer interface intended to handle both the complex data structures and tie together the disparate processors that make up OntoSem. In other words, the DekadeAPI is a class library, providing top-level functionality to the programmer, allowing various researchers to access the data and processors that form OntoSem. Most importantly, a strong API allows for interaction between the various data resources that was previously not available.

With the construction of a strong library of functions, the architecture of the DII system could now be realized. In order to assure use of the most up-to-date version of the OntoSem analyzer, as well as a view of the latest static knowledge, users of the environment would need to periodically connect with the main development server. However, the majority of interaction and display could be left to the client system, so a

client/server model was a natural fit for the DII system. This meant that two separate sub-systems would be built, the DIIServer, and the DIIClient.

The DIIServer was created to use the DekadeAPI to handle user requests to OntoSem, and to construct current views of the static knowledge resources from the database. The DIIServer was installed on the development system, and set to listen for incoming messages on a certain open port. When a message is received, the DIIServer interprets the command, translates it into DekadeAPI function calls, and returns the result back to the requesting client.

The DIIClient (also referred to as just DII) was created as a graphical user interface (GUI) wrapper around the DekadeAPI. DII would make function calls into the DekadeAPI, some of which would be interpreted locally, while others would be transmitted to the DIIServer, which would then respond appropriately to the request. With this architectural philosophy, the DII system would allow for simpler interface programming by users desiring a certain view into the OntoSem system and related static knowledge resources.

This paper is organized in the following manner: in Chapter 2, there will be a discussion on the background of ontological semantics, the driving theory behind the OntoSem analyzer. Chapter 3 will introduce the OntoSem analyzer, and touch on each of its main processors and static knowledge resources. In Chapter 4 motivation will be provided for the need of a standalone development, acquisition and demonstration environment for OntoSem. Chapter 5 will discuss other related works in knowledge acquisition environments. Chapter 6 will introduce and discuss in detail the DekadeAPI. In Chapter 7 there will be a discussion on the DIIClient, as well as a touch on its construction and design choices. Chapters 8, 9, 10 and 11 will discuss the default interfaces included in the DIIClient (these being the OntoSem analyzer demonstration environment, as well as the lexicon, ontology, and fact repository editing environments). Current uses of the DII system, future work and conclusions will be presented in Chapter 12.

2. Ontological Semantics

Ontological semantics is a theory for natural language processing that relies heavily on a structured "world view" model, the ontology, to help construct meaning out of text [7]. In practice, ontological semantics creates semantic dependency structure of the source text by deriving and disambiguating the lexical meanings of individual words and phrases, and then combining them in a manner prescribed in a pre-defined structure.

An ontology, in this framework, is a collection of objects organized in a treelike structure, with children in the tree inheriting properties from their parents. Each object in the tree represents a real-world concept, and is defined in terms of other concepts in the ontology. A *HUMAN*, for example, could be defined as being a child concept of *MAMMAL*, inheriting the properties associated with it. *HUMAN*, however, could further expand or redefine the properties granted to it by default. For instance, a *HUMAN* could take part in some event, such as a *INHERIT*. Although it is conceivably possible, it is not

likely that a non-human mammal will frequently be involved in receiving inheritance. In this manner, a view of the world is constructed into an ontological object.

Ontological semantics uses this worldview to both disambiguate and append meaning to an analysis of a text. Every lexical entry in such a system defines which ontological concept it maps to, and how it is structurally used. When a word is uncovered in analysis that has multiple meanings, the different ontological mappings can assist in disambiguation. After the true meaning of a word is identified, meaning can be added to the analysis of the text by searching the properties and default values of the mapped concept, as well as any inherited values.

Ontological semantics provides for a robust descriptive worldview meta-language, that is language independent. Text analysis in this manner allows for rich meaning to be extracted and presented in an intuitive way.

3. OntoSem

The OntoSem analyzer is a collection of text processing tools and static knowledge resources designed to follow the theory of ontological semantics. When used in harmony with each other, the sub-processes and data contained in OntoSem can produce an ontologically based text meaning representation (TMR) from any input text. Specifically, OntoSem is constructed of four primary processors, the Preprocessor, the Syntactic Analyzer, the Semantic Analyzer, and the Semantic Reference Analyzer. Additionally, OntoSem is comprised of four main static knowledge resources, the lexicon, the ontology, the onomasticon, and the fact repository. In this section a discussion of each of these processors and resources follows, beginning with the static knowledge resources.

3.1 The lexicon

The lexicon is a static knowledge resource, which can easily be likened to a dictionary. It contains a vast list of language dependent words, each of which is marked up with a structure of supporting and descriptive data. For instance, each lexicon entry must contain a value for its part of speech (noun, verb, etc.). Lexicon entries also contain syntactic and semantic structures, or templates, which define how the entry can be properly used in a sentence. Because words may have multiple meanings, every lexicon entry must be uniquely identified with a sense tag. A sample lexicon entry is shown in table 3.1.

3.2 The ontology

The values found in the sem-struc field of the lexicon entry shown in table 3.1 are derived from the second major static knowledge resource, the ontology. The ontology used by OntoSem appears as a mostly treelike structure, with the root node labeled *ALL*.

Entry	Value
name	absinthe
sense	absinthe-n1
cat	n
synonyms	
hyponyms	
def	A green liqueur flavored with wormwood or anise and other herbs.
ex	
comments	
syn-struc	((root $var0) (cat n))
sem-struc	(liquor (has-object-as-part wormwood) (color green) (has-object-as-part anise))
meaning-procedure	
morph	
abbrev	
output-syntax	
tmr-head	
lastupdated	2006-01-03 00:00:00

Table 3.1: A sample lexicon entry, "absinthe".

ALL has three children, *OBJECT*, *EVENT*, and *PROPERTY*. From these three nodes, the rest of the ontology unfolds. Objects and events reference properties in the ontology, and have certain values assigned to them (these values could be other objects and events, numeric values, etc.). For example, the ontological concept *LIQUOR* is displayed below, in Figure 3.1.

SLOT		FACET	FILLER
MADE-OF	edit	SEM	BARLEY, RYE, CORN, ...
IS-A	edit	VALUE	ALCOHOLIC-BEVERAGE
COLOR	edit	SEM	TAN, BROWN
TIMESTAMP	edit	SEM	Modified Mon, Jun 28, 2004 by marge;Modified Tue, Oct 7, 2003 by inna;Modified Mon, Oct 6, 2003 by inna
DEFINITION	edit	VALUE	a strong alcoholic liquor distilled from the fermented mash of grain

Figure 3.1: The ontological concept *LIQUOR*.

One can see that the concept *LIQUOR* is a child of *ALCHOLIC-BEVERAGE*, that it can be made of *BARLEY*, *RYE*, or *CORN*, and that it's *COLOR* can be *TAN*, or *BROWN*. Cleverly, one can define the lexical entry absinthe (as shown in Table 3.1) to be a *LIQUOR*, but also to have a *COLOR* that is *GREEN* and to have the additional property *HAVE-OBJECT-AS-PART* be defined as *WORMWOOD*. In this way, there already is an interaction between these two static knowledge resources: at the very least the lexicon depends on the ontology. This interaction between static knowledge will be revisited in Chapter 4.

3.3 The onomasticon

The onomasticon can be considered a specialized version of the lexicon. The onomasticon is list of proper nouns (names, places, etc.) with mappings to the type of data that they are often considered. The onomasticon is the proper place for the word

"George". It does not refer to a specific George, but rather would list George as being a male name.

3.4 The fact repository

The fact repository holds all information concerning specific instances of ontological concepts. When a text is analyzed, and the word "absinthe" is discovered, it is mapped to the ontological concept *LIQUOR*, as shown above. For the duration of the analysis of the text, this instance of the word "absinthe" will be given a unique identifier, such as *LIQUOR-1*. This identifier points to an instance, or a temporary copy, of the concept *LIQUOR*. This instance can be further modified beyond the default values found in both the concept and in the lexicon entry, by analyzing the surrounding text. For instance, the sentence, "This absinthe is blue in color." would result in an instance of *LIQUOR* whose *COLOR* property was *BLUE* instead of the *TAN*, *BROWN*, or *GREEN* that it could have been. If this instance (*LIQUOR-1*) is deemed important enough, it can be stored for future reference – in the fact repository. In other words, the fact repository is the location where instances of ontological concepts, which are often modified beyond their default values, are stored.

The fact repository is frequently used for collecting world knowledge data, such as information concerning the President of the United States. As an example, the fact repository could contain the fact *HUMAN-231*. This *HUMAN* instance could have the property: *HAS-NAME* George Bush. The fact could then be supplemented with further information, which could be anything that is a valid concept in the ontology (*AGE*, *AGENT-OF*, *HAS-OBJECT-AS-PART*, etc.). Here again once can see another example of interconnectivity between static knowledge resources in OntoSem.

3.5 The Preprocessor

The preprocessor is the first analyzer that a text must pass through before a TMR can be produced. The preprocessor interacts with the lexicon and onomasticon to tag each word in a text with all of its valid parts of speech. When the word "man" is discovered in the text, referencing the lexicon will show that this word can be both a noun, and a verb. The word "George" will be found in the onomasticon, and thus tagged as a noun.

3.6 The Syntactic Analyzer

The results of the preprocessor can be fed into the syntactic analyzer. This processor is responsible for uncovering all syntactic structure in a text. Words can be combined to form noun phrases, prepositional phrases, clauses, etc. By previously identifying the part of speech for each word, the syntactic structure of the text can be uncovered.

3.7 The Semantic Analyzer

With the syntactic structure deciphered, the real work can begin. Using the ontology as a reference, the semantic analyzer can construct a TMR from the sentence. Mapping

lexical entries to their corresponding ontological entries, and constructing details about each instance found in this way. The result is a structured text meaning representation of the input, a collection of instances of the concepts found – each with details specific to the text found within.

3.8 The Semantic Reference Analyzer

Using the final static knowledge resource, the semantic reference analyzer looks through the fact repository to try to match existing knowledge concerning a stored instance with the knowledge found in the text. Additionally, the semantic reference analyzer handles cross-sentence pronoun resolution, as well as other forms of reference ambiguity. An example of cross-sentence pronoun resolution would be determining who the "he" refers to in the following passage:

Jim went to the store to buy groceries earlier. He did not find any cheddar cheese, however.

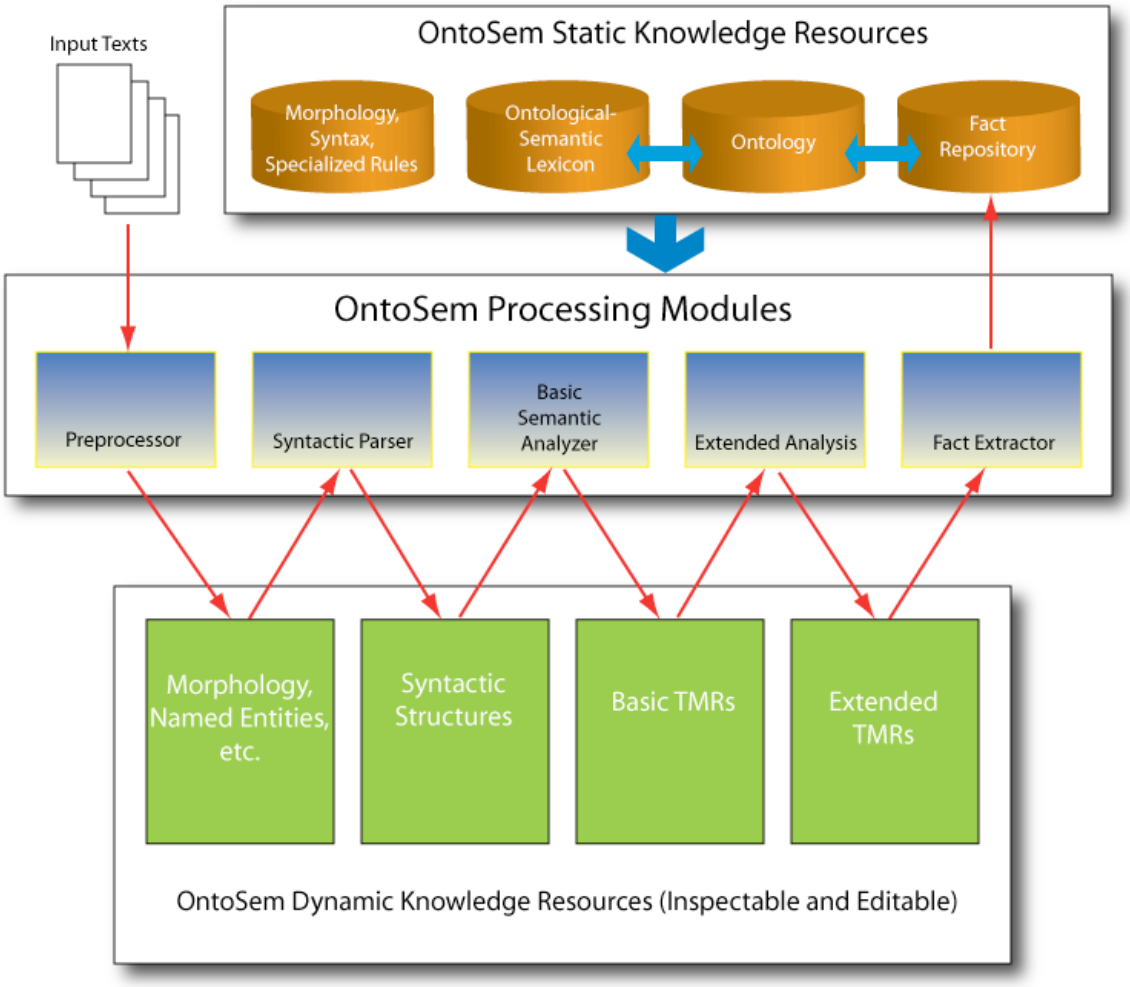

Figure 3.2: The OntoSem Process.

Identifying that “he” is an extension of “Jim” is imperative to properly producing a TMR of the entire text. A global view of the OntoSem process is presented in Figure 3.2.

4. Motivation

The previous section discussed the various components that make up the OntoSem analyzer. It is clear that each of the processors accesses at least one of the static knowledge resources during its execution, and several of the static knowledge resources require other resources in order to be properly formed and to maximize the expressive potential of the system. Certainly, having a single development environment that encompasses all of these data types and processors would yield benefits that could not be found in a collection of separate environments. However, this is not the only motivation behind the creation of the DII system. As will be seen in this section, creating a single environment would not only facilitate the production of higher quality static knowledge, but would also allow for easier debugging of the analyzer – both between stages of analysis, as well as between analysis and a static knowledge resource.

4.1 Knowledge acquisition made easier

Acquiring knowledge for a natural language processing system is often the most labor intensive and time consuming phase. Further, in any evolving NLP system, it is a phase that never ends. To make an environment that facilitates knowledge acquisition, it should allow the users to enter data in a familiar, and natural manner. It can then be translated automatically into any form that is most appropriate for the end level application. It would be cumbersome and silly to force users to enter ontological concepts directly to a database via SQL (structured query language); manually entering each property and value of a concept would be time consuming, and prone to error. Alternatively, a simple form could be constructed that would allow data to be entered into a structure, which at the click of a button is converted automatically into a series of SQL statements and executed.

Validation is a problem that is equally addressed in this manner. If every lexicon entry had to be crosschecked by hand for perfect structure and valid field data, it would break the flow of knowledge acquisition from the user. Imagine, as a metaphor, the difference between automatic spelling and grammar checking in a word processor, versus a simple typing application with no such features. In the latter case, the user must return to the document after scripting it, to verify its accuracy – a task that is both tedious and inconvenient.

Continuing with the word processor metaphor, the solution was to give the processor access to a dictionary of valid words. When a word is encountered that is not found in the dictionary, the user is immediately identified. In a natural language processing development environment, a similar method can be employed. By giving the lexicon acquisition tool (the word processor) access to the current ontological data (the dictionary), an acquisition expert could be immediately notified when a referenced ontological concept was not valid (for example, it does not exist).

4.2 Demonstration and debugging

Each entry of static knowledge found in OntoSem can be likened to a class data type in object-oriented programming. Its use is self-defined, and it can be instantiated any number of times during the course of execution. Because of this, it can often be difficult to tell whether the incorrect results obtained from the analysis of a text are the result of bad code in the analyzer, or bad structured knowledge in the ontology or lexicon. If the data in the static knowledge resources are not available to a program that assists in the execution of the analyzer, then the work of tracking down a bug in the system is greatly increased. Alternatively, by allowing the same functions that run the analyzer and receive it's output to access the static knowledge, the user would be able to easily compare the results of the analysis to the original static knowledge to check for obvious errors.

Further, a system that ties together each of the four main processors of OntoSem could expedite demonstration of the system as a whole by fluidly accessing each processor in turn, allowing for the user to simply issue a single command in order to see the resulting TMR. Because the system is capturing the output of OntoSem at each main phase, it is also capable of displaying these intermittent steps in a user-friendly environment. A user would be capable of modifying these values before passing them on to the next level of processing. The process of manually modifying the output to produce a "correct" TMR (a Golden TMR) could easily be appended to an environment that already was capable of tying together all the pieces that create OntoSem.

The motivation for creating an all-encompassing environment for the development and evaluation of OntoSem should be clear:

1. Knowledge acquisition and validation will be facilitated by a connection between static knowledge sets.
2. Error chasing will be expedited by connecting the static knowledge with the processors in a user-friendly environment.

In the next section there will be a discussion on the existing implementations for various NLP and knowledge acquisition systems, that are both all-encompassing and separated applications, and highlight what functionality is useful for inclusion in an environment developed for OntoSem, and what functionality does not apply.

5. Prior Work

The following section is a discussion of the various works done by other researchers related to knowledge acquisition interfaces and NLP environments, touching on five prior works done in this area, including one developed as a precursor to DII, to specifically support the OntoSem analyzer.

The Protégé environment for knowledge based systems development, created by Gennaro et al. at Stanford University [1] has undergone four distinct evolutions to produce the current version, Protégé 2000. Protégé was developed as a tool to allow domain experts to create knowledge base acquisition tools for use by knowledge engineers. The tool was designed over time to support an ontological structure of knowledge construction through custom designed application specific interfaces (using an API created for the project). In many ways this distinctly reflects the knowledge acquisition requirements of DII. Specifically, each of the static knowledge resources needed by OntoSem requires a user friendly, custom created acquisition interface. Many of the decisions made in the DII system design show in the Protégé design as well: plug and play custom forms, cross-platform Java architecture, graphical viewers, and as of Protégé 2000, the ability to modify any part of the knowledge in a non top-down manner.

Another work whose motivation is similar to DII is the Berkeley FrameNet Project [2]. The FrameNet environment is a suite of web-based interfaces supporting the creation of a semantic database through human authoring translated into a machine-readable form. The project includes a few subsets of data, notably a lexicon and a frame database (similar in function to the ontology structure used by OntoSem). A specific methodology is used for acquiring the data, and validation between data sets is also done (this is conceptually similar to verifying that concepts used in the lexicon are thoroughly defined in the ontology). The FrameNet project involves a series of independently developed tools that are stitched together through a web interface, in much the same way the original DEKADE environment was.

Another system that closely mirrors the philosophy of the DII design is the Annotation Graph Toolkit (AGTK) [3] designed at UPenn. AGTK made use of a similar architectural layout, with the client side interfaces interacting with the local data, and the local data being stored in a larger repository server-side. The project also supports an API for custom user interfaces. The TreeTrans project [4] also developed as part of the AGTK, involves graphical displays for viewing and manipulating tree structures: a primary necessity for DII.

Further challenges of developing a semantic ontology and integrating an NLP parser are reported on in [5][6], reports on ConceptNet, developed at MIT. Similar issues are discussed, such as relevant ways to allow the user to input flexible data without having to enter it in a machine-readable form. The structure of an ontological entry discussed is similar to the structure used by OntoSem. ConceptNet was initially populated using a knowledge acquisition questionnaire that was posted on the Open Mind Common Sense Web site [8]. This site asked users to enter answers to simple questions, such as “A knife is used for…”. This interface allowed for common sense knowledge to be acquired into a system without the need of domain or knowledge acquisition experts.

Finally, the DEKADE system developed at UMBC [12] (which is a precursor to the DII system), was designed as a web-based interface into the various processors and knowledge resources that comprise OntoSem. Harmonious integration between static knowledge was achieved by allowing the web site universal access to all databases that

OntoSem can reference. The system supported a view into the analysis of a sentence at each processor, along with editing features. Additionally, the system allowed for lexicon browsing and modification, as well as ontology browsing. Although the system mostly succeeded in meeting the requirements for an integrated environment, it failed to do so with grace or efficiency, often suffering from connectivity and speed issues.

Table 5.1 shows a cross examination of each of the discussed systems, comparing features and capabilities that are found, are fundamental, or are not available to the DII architecture.

	DII	Protégé	FrameNet	AGTK	ConceptNet	DEKADE
OntoSem support	Yes	No	No	No	No	Yes
Globally Accessible Data	Yes	Yes	Yes	Yes	Yes	Yes
Special Software Required[1]	Yes	Yes	No	Yes	No	Yes
User API Provided	Yes	Yes	No	Yes	Yes	Yes
Web-based GUI	No	No	Yes	No	Yes	Yes
Access to up-to-date Knowledge	No[2]	Yes	Yes	Yes	Yes	Yes
Knowledge Editing Tools	Yes	Yes	Yes	Yes	Yes	Some

Table 5.1: Comparison of existing system to the DII design.

In short, a variety of toolkits for various ontological or NLP related systems have been constructed, all with a specific purpose in mind. DII has also been designed in this way; DII hopes to bring the development and demonstration of the various tools and data structures required of OntoSem easily into the experts hands.

6. DekadeAPI

To properly address all of the requirements of a development and demonstration environment for the OntoSem analyzer, strong backend architecture would need to be developed first. This architecture would need to have access to each static knowledge resource, and each processor in the OntoSem analyzer, and would need to handle systems-level operations such as data manipulation and storage, as well as network

[1] It is assumed that all users have access to a web browser, and the Java Virtual Machine.

[2] The user must request the knowledge, it is not presented by default.

communication. The details of how the data is organized and stored on a drive, or how the command to analyze a sentence is communicated to the server should be shielded from the end-developers eyes. In this manner, the DekadeAPI was created.

The DekadeAPI is built on Java 1.5 technology, using a PostgreSQL database, and contains a wide array of OntoSem specific tools as well as several generally useful functions for handling low-level operations. The package structure of the DekadeAPI is shown in Figure 6.1.

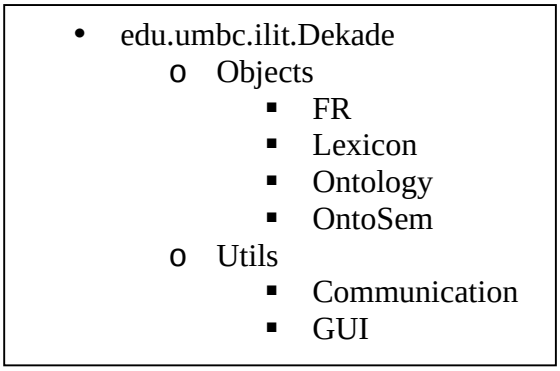

Figure 6.1: DekadeAPI structure.

To handle all of the data interaction, the edu.umbc.ilit.Dekade.Objects package was created in the DekadeAPI. Inside this package a distinct architecture for each of the static knowledge resources lexicon, ontology, and fact repository, was created. Each architecture was constructed from the ground up, by analyzing what the most atomic form of organizable data in that resource was, and expanding from there (with careful thought given to efficiency of data access, and ease of data manipulation).

6.1 edu.umbc.ilit.Dekade.Objects.Lexicon

The data structure that holds and manipulates the lexicon while in the DII environment consists of three primary objects. At the lowest level is the *LexiconEntry*, a class that contains the values associated with each property of one lexicon entry (see Table 3.1). The class also contains a timestamp for recording when the last modification to the entry was made. This class could be considered a single record or entry in a dictionary, containing the knowledge of that entry, with no regard to the data surrounding it. At a slightly finer grained level, lexicon entries whose name begins with a certain letter are grouped into an object called the *LexiconLetterList*. This object keeps and maintains a sorted list of entries that are all lexicographically similar. At the highest level is the *LexiconObject*, a class that catalogues and references 27 distinct letter-lists (an extra one for entries who do not start with an alphabetic character), and provides a variety of easy access functions into the data contained within. For example, a user of the DekadeAPI would only need to make a call to the lexicon object's *getLexiconEntryBySense* function to retrieve the structured entry whose name matches the parameter – in other words the lexicon object handles parsing the parameter to determine which letter list to query, and the letter list handles efficiently searching for the entry and retrieving it.

In addition to these data storage and control classes, the package also provides a simple load and save functionality, as well as the ability to commit an entry to the server's data repository, and to retrieve new or updated data from the server.

6.2 edu.umbc.ilit.Dekade.Objects.Ontology

The ontology data structure follows a similar pattern to that of the lexicon, with one subtle difference. The lowest level of ontology data storage is the *OntologyProperty*, a class who handles exactly one row of information about a single ontological concept. This class stores the values of a single property, with no reference to which concept this property belongs. One level up in the design is the *OntologyEntry*, which contains a collection of properties as well as a timestamp of the last modification to the structure. Further up in the architecture is the *OntologyLetterList* whose function is identical to the *LexiconLetterList*: it keeps and maintains an organized list of ontological concepts whose names are alphabetically similar. The highest level in the ontological structure is the *OntologyObject* which is responsible for keeping a tab on all of the letter lists, as well as keeping a unified view of the ontological tree structure (as defined by the values in the concepts).

Similarly, the ontology package also provides the same basic low-level functionality to the environment programmer: the ability to load and save the data locally, as well as to commit data to the server, and read new and updated data from the server.

6.3 edu.umbc.ilit.Dekade.Objects.FR

The fact repository is organized in a slightly different method from the lexicon and ontology, which stems from its unique use and organization in the DII environment. To facilitate and encourage research using the OntoSem analyzer, the DII system employs a method to automatically construct and organize fact repository knowledge from analyzed texts. In order to maintain some sanity in the fact repository, a package of FR functions was integrated into both OntoSem and the DIIServer (this is discussed in detail in Section 6.6), and introduced the concept of the *subfr*. A *subfr* is a subset of the main FR, whose elements are distinct and independent from others in the FR. This means that a fact in one subfr can have the same name (and different or identical properties) as another fact in a different subfr – all while being housed in the same FR. The data storage introduced in the DekadeAPI reflects this organization. At the bottom level is the *FRProperty*, a class nearly identical in structure and design to the *OntologyProperty*. Containing a list of the properties is the *FRInstance* class – an example of a single instance in the FR. Instances are contained in a dynamically sorted object called the *FRSubfr* class. This class reflects the contents of a subfr found on the server, and is distinct from any other subfr.

The DekadeAPI FR package also provides the same functionality as the other data management packages.

6.4 edu.umbc.ilit.Dekade.Objects.OntoSem

To support the debugging and demonstration of the OntoSem analyzer in action, a structured object for managing the values returned by the various processors was created. At the lowest level, the *Sentence* object reflects a single sentence in an analyzed text. This object contains the values of the original text, along with the analysis of the text at the preprocessor, syntax, semantics, and reference stage. Additionally, as the user can modify the output of each of these values, and continue with the analysis, the sentence object handles a list of the analysis results at each level of modification. A collection of sentence objects makes up the *AnalyzedTexts* object, which represents the results of analysis of an entire input text.

6.5 edu.umbc.ilit.Dekade.Utils

Contained within various packages inside the Utils folder of the DekadeAPI are a series of useful and necessary objects that facilitate the development of the DII environment. Most importantly, the *DekadeMessage* object, found in the Communications folder, is the only data which is allowed to be sent to the DIIServer. The message contains a variety of flags and command strings that allow the user to request data and operations from the server, and allow the user to receive information in return. The use of the *DekadeMessage* class is integral to the client/server architecture of the DII system.

6.6 The FRAPI

Although not specifically included in the DekadeAPI class architecture, the FRAPI was developed in conjunction with the DekadeAPI, and supports server-side access to the FR database through a series of carefully structured SQL statements. The FRAPI introduced and maintains the notion of the *subfr*, which was touched on in Section 6.3. The need for the subfr arose from the desire to facilitate and encourage experimentation and research in various areas using the OntoSem analyzer. In a real world environment, having the voluminous amounts of data produced by OntoSem stored in a single FR would make for nightmarish organizational problems. The goal was to alleviate these issues by allowing each independent research project (and even each independent test run within a research project) to have separate carved out FR space. This would allow for simple keyword filtering, as well as easy database maintenance (old project data could be completely removed without fear of harming current data).

The FRAPI provides a strong background to organizing the FR data on the server side. In order to write, modify or access data, both the desired FR, and subfr must be provided. In addition, a variety of commands to simplify searching and cross searching the data space were provided to remove the need to create SQL statements by the end user. A data flow diagram of FRAPI interactions is presented in Figure 6.2.

With the DekadeAPI and supporting FRAPI as solid support architecture in place, in the next chapter there will be a discussion on the system design of the DIIClient.

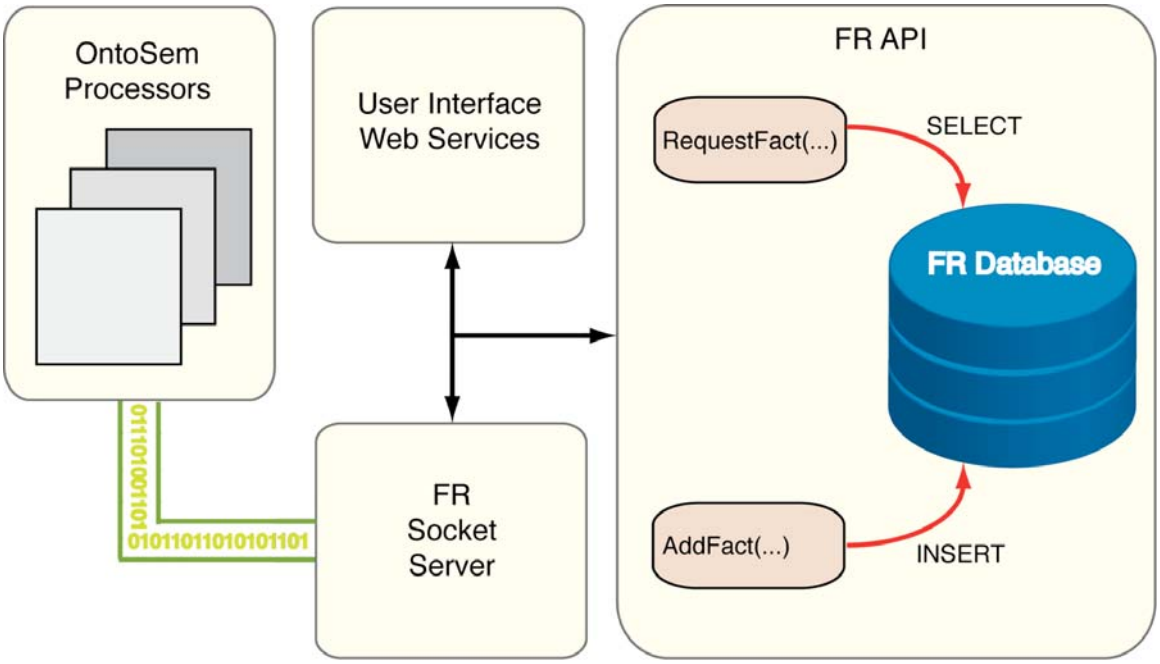

Figure 6.2: Data flow around the FRAPI.

7. DIIClient

With a solid foundation of classes constructed, a flexible GUI to facilitate acquisition and demonstration could now be constructed. It was necessary to address certain concerns when constructing the interface, which are enumerated below:

1. The client must be cross-platform.
2. The client must be lightweight.
3. The client must be easily extended and enhanced by other developers.
4. The client must remember its state from function to function.
5. The client must minimize network traffic.
6. The client must keep synchronized data.

7.1 Cross-platform

The client program would need to be capable of running on any reasonably expectable platform an end user could select. The program could not be developed exclusively for a Windows or Linux environment, but should be able to cater to Windows, OS X, and Linux users. Further the look, feel, and most importantly, function, of the program must be identical across these platforms.

7.2 Lightweight

The client program should take up as small a memory space as possible during the course of execution, at the discretion of the user. Due to the tremendous amount of data found in each of the static knowledge resources, loading each one into memory at the start of the program would not only sap system resources, but would also cause a slow load time.

Instead, the program should not load any static knowledge until requested, but after which it should not remove the data from memory to maximize access time of the data throughout the course of execution.

7.3 Extendable

The client program should be capable of accepting add-on user interfaces developed with the DekadeAPI without the need to recompile or redistribute the existing executable. In other words, the client program should support some sort of drag and drop, or "plug-in" architecture, allowing independent developers with access to the DekadeAPI the ability to construct interfaces that are automatically detected and added to the client program when installed on a users machine.

7.4 State

The client program must be able to toggle between interfaces and not forgot the state it was in when it left the first one. As an example, if a user was developing a lexicon entry in an interface, and decided to toggle to another interface to reference an ontological entry, returning to the lexicon interface should find any unsaved work exactly as it was prior to swapping views. This is contrary to the natural operation of a web-browser (although a browser can be made to recall these values by setting session variables and refreshing a page with them).

7.5 Minimize Traffic

The client must not make a request to the server for every action taken. Only requests that require the specific services the server can provide should be made. The server should be limited to distributing data in chunks (as opposed to constant streams), and analyzing texts using OntoSem; this should be done both for efficiency (as OntoSem can take some time to run), as well as to allow the client access to the latest version of OntoSem.

7.6 Synchronization

Perhaps the most important feature of an acquisition interface is the capacity to keep data synchronized across all users. The problem, in distributed computing, is often called the write-after-write (WAW) scenario. In the WAW scenario, two (or more) users request data from a server to their personal machines. Each user, without knowledge of the others actions, edits the data independently. The first user submits the data back to the server, so that future requests by other users will be able to see the work done. The second user, unaware of this activity, proceeds to submit his data to the server as well. The second user's data overwrites the first user's data, and work is lost. As a large part of the DII system is knowledge acquisition, this scenario is unacceptable, and must be avoided at all costs.

7.7 Solution

The appropriate solution was to create a GUI developed in Java/Swing (Swing is a graphic component library developed in Java). By developing the system in Java, the first concern, cross-platform usability, would be easily solved. Any user having a Java Virtual Machine (JVM) installed would be capable of executing the client. To handle the second concern, the client would not force the static knowledge to be loaded at the time of execution. Instead it would be up to the developer of any interface to supply an easy-to-access function (likely a button) to load the relevant data into memory – functionality that is provided by the DekadeAPI.

To allow the DIIClient to be extendable, it would need to have no interface programming associated directly with the main code. Instead, the client was designed to search a specific *./lib/* folder contained inside the main DIIClient folder for properly formatted .jar files. A .jar file is essentially a library of compiled Java code, compressed into a zipped format. The DIIClient could find each .jar file inside the folder, extract its contents, uncover the package and class content, and instantiate an object found within. If the object is properly constructed, it will be an extension of the *JPanel* class, an object that is found within the Swing framework. This object can be added as a tabbed panel to the DIIClient. In this method, any developer with access to the DekadeAPI, and a very short tutorial, could produce a custom interface that could simply be dropped into any users library folder, and would show up in the DIIClient. This design closely mirrors the design of the Protégé environment. In both cases, the interface programmer must follow a small series of requirements to make the plug-in detectable, and has free range over the supporting APIs.

Using a tabbed panel approach also solved the fourth concern. The natural behavior of a tabbed panel in the Java/Swing environment is to render the request panel onto the screen, and to not render all other panels associated with the tabs. However, the act of not rendering a panel does not remove the panel from memory, or in any way invalidate the data it contains. By selecting a different panel from the currently active one, a user is choosing to hide the original panel from view. A simple metaphor is to image a spiral notebook. Writing on one page, and then flipping to a new blank one does not erase the contents of the first. To access them, one need only flip back to the original page. In this manner the DIIClient would be able to retain the unsaved information in one editor while a user referenced another editor or browser for clarifications.

To minimize network traffic, the fifth concern, the client would store all static knowledge locally, and perform all rendering and interface interaction on the local machine. Network connectivity would, therefore, only be used when the user requested updates to the static knowledge, or requested to update the server's static knowledge, or requested the analysis of some or all of a text by OntoSem. To maintain synchronization (the final concern), the client would need to use the already existing timestamps found in each of the objects in the DekadeAPI. When the client attempts to transfer an edited knowledge entry, the server could test the timestamp with the timestamp in the database. If the client's timestamp was found to be the same as the timestamp in the database, then the

WAW scenario does not apply, and the data can be updated (along with both timestamps). If, however, the client's timestamp is found to be older than the one in the database, the client has been working with data that has since been modified by another user, and hence the WAW scenario does apply, the data is not updated, and the user is made aware of the situation.

Using this client/server Java based architecture; the DIIClient was constructed to handle all of the concerns that must be addressed by a valid acquisition and demonstration environment for the OntoSem analyzer. Figure 7.1 details the flow of control for the DII system. The next four sections will discuss the four default DII interfaces: the lexicon editor, the ontology editor, the fact repository editor, and the OntoSem text analysis browser.

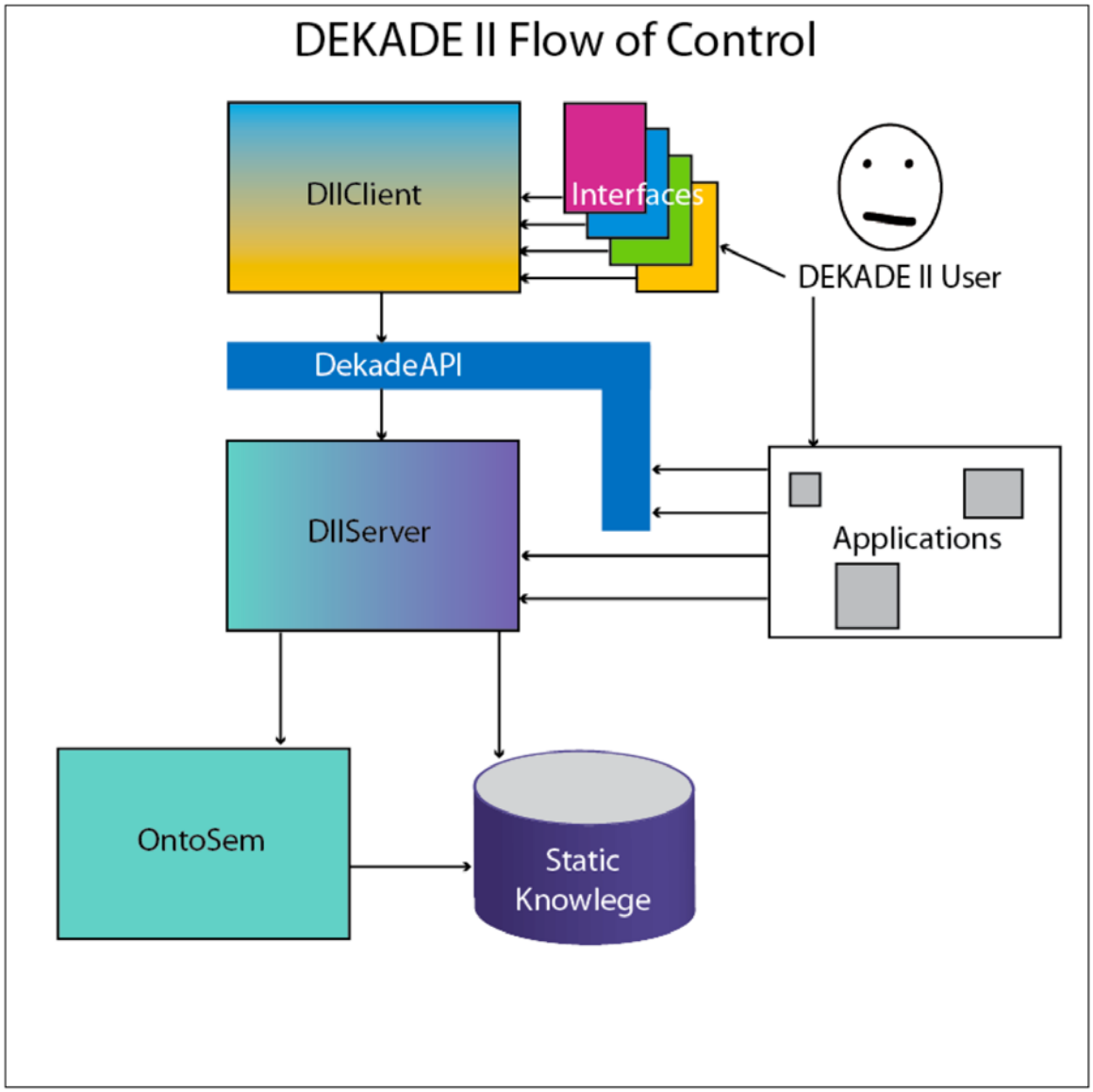

Figure 7.1: Flow of control in the DII system.

8. The Lexicon Interface

The lexicon interface was developed using the DekadeAPI and in accord with the DIIClient interface requirements. The interface was designed to accommodate the arduous task of acquiring volumes of lexical data. The intuition behind the design was a smooth flow of control, starting with selecting an existing word or creating a new one, and moving through the design process to a completed entry. Figure 8.1 shows a screenshot of the lexicon interface.

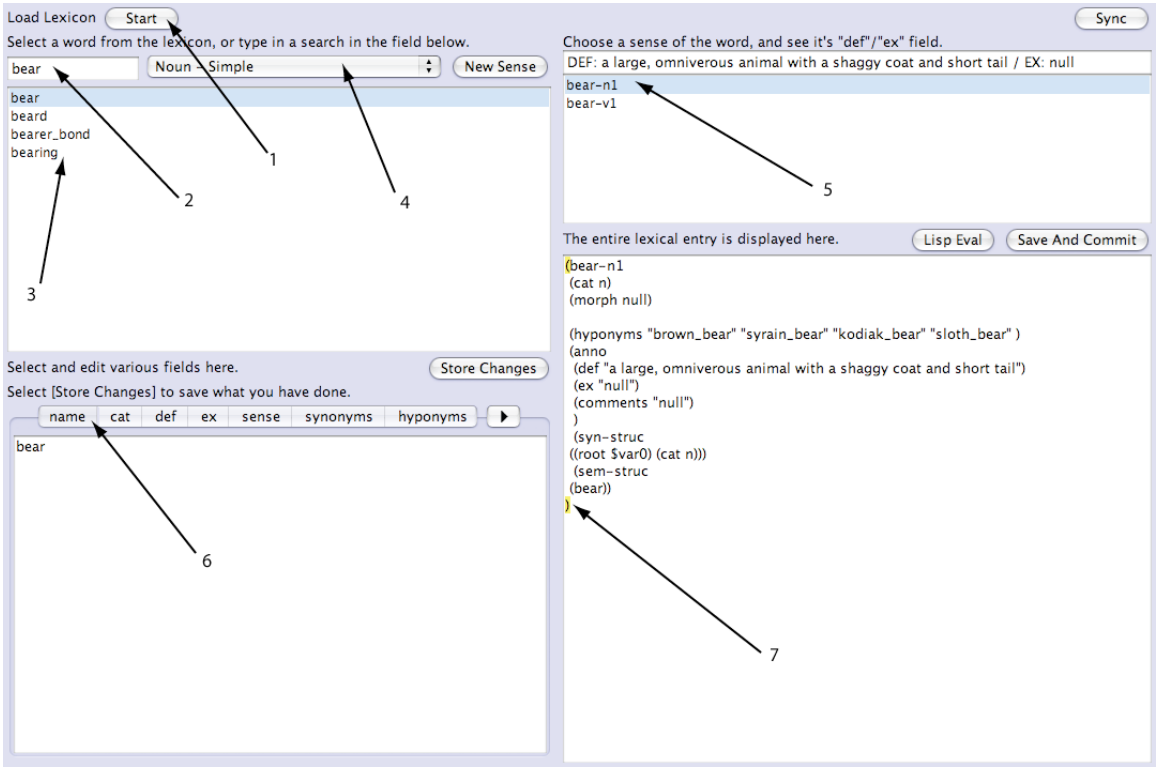

Figure 8.1: The Lexicon Interface.

As specified by the design constraints of the DIIClient, the lexicon is not loaded into memory automatically by the program, instead a button has been provided to do so inside the interface {Fig:8.1.1}. Once the lexicon has been loaded into memory, the user can search the word bank by typing into a text box {Fig:8.1.2}, each letter that is entered filters out non-matching words. All matches to the users query are shown in a list directly below the search box {Fig:8.1.3}. A word that perfectly matches the query in the search box will be automatically selected, however a user can select any word available instead. If the desired word is not available, a user may opt to create a new sense of the word, and can select from a drop list of predefined templates {Fig:8.1.4} to assist in faster acquisition. After a word has been selected, all matching senses of the word (recall a word can have more than one meaning) are displayed in the sense list box on the right side of the screen {Fig:8.1.5}. The user is free to select any sense they desire; however the first one will be automatically selected (as many words only have one sense this tends to eliminate an extra step on the user's part). When a sense has been selected, the remainder of the screen is populated. In the lower left, a tabbed pane of each property of the lexicon entry is displayed {Fig:8.1.6}. The user can select any property and modify

the original contents; the changes are displayed in the fully formatted output on the right side of the screen. If the user prefers to edit the entry as a whole, the right side view is also editable, and provides automatic parenthesis highlighting {Fig:8.1.7} to assist the user.

Additionally, the interface also provides a direct way to test the entry for validity, as well as download new updates and commit changes made to an entry. The flow of control moves in a natural way from top left, to bottom right, and seeks to minimize the required tedious interaction between the user and the interface.

9. The Ontology Interface

Like the lexicon interface, the ontology interface was developed following the requirements of interface construction for the DIIClient application. The ontology interface, unlike the lexicon interface, was designed primarily as a browser, with editing functionalities hidden from the main screen view. The design emphasizes the structure of the ontology, as well as each individual concept within. Figure 9.1 shows the main ontology interface panel.

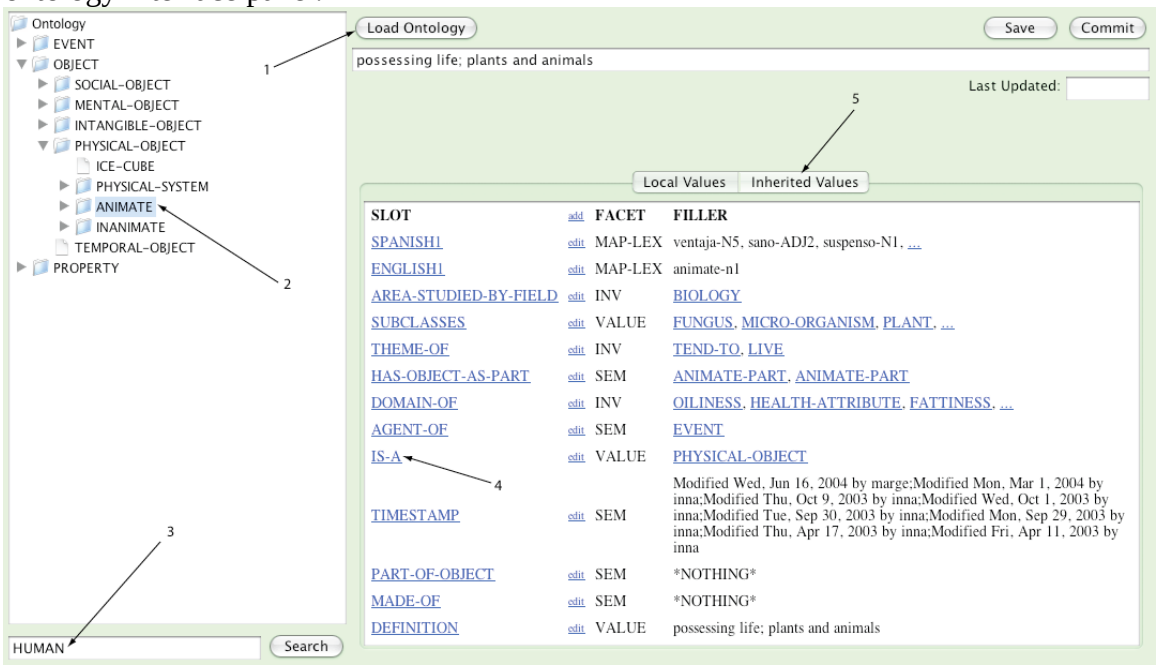

Figure 9.1: The Ontology Interface.

Following the lightweight requirements of the DIIClient architecture, the ontology is not loaded automatically, instead the user is presented with the option to load the data {Fig:9.1.1}. Once the ontology has been loaded into memory, a collapsed tree structure is displayed on the left side of the screen. The user is free to browse through the ontology in a logical manner. When the user finds the desired concept, clicking on the text in the tree {Fig:9.1.2} will produce a detailed view of the concept on the right side of the screen. If the user had known the exact concept name, a search bar for automatic selection of the concept is provided {Fig:9.1.3}. When investigating the properties of the ontological concept, any property value that is itself a concept is highlighted {Fig:9.1.4},

and by selecting it the user is automatically brought to that concept in the ontology. As a key aspect of the ontology is inheritance, the user can select a tab to view all the data a concept inherits from other concepts further up in the tree {Fig:9.1.5}.

If the user desires to edit a slot in the ontology, selecting the small edit link next to the slot name will produce an editor dialog. This dialog is shown in Figure 9.2.

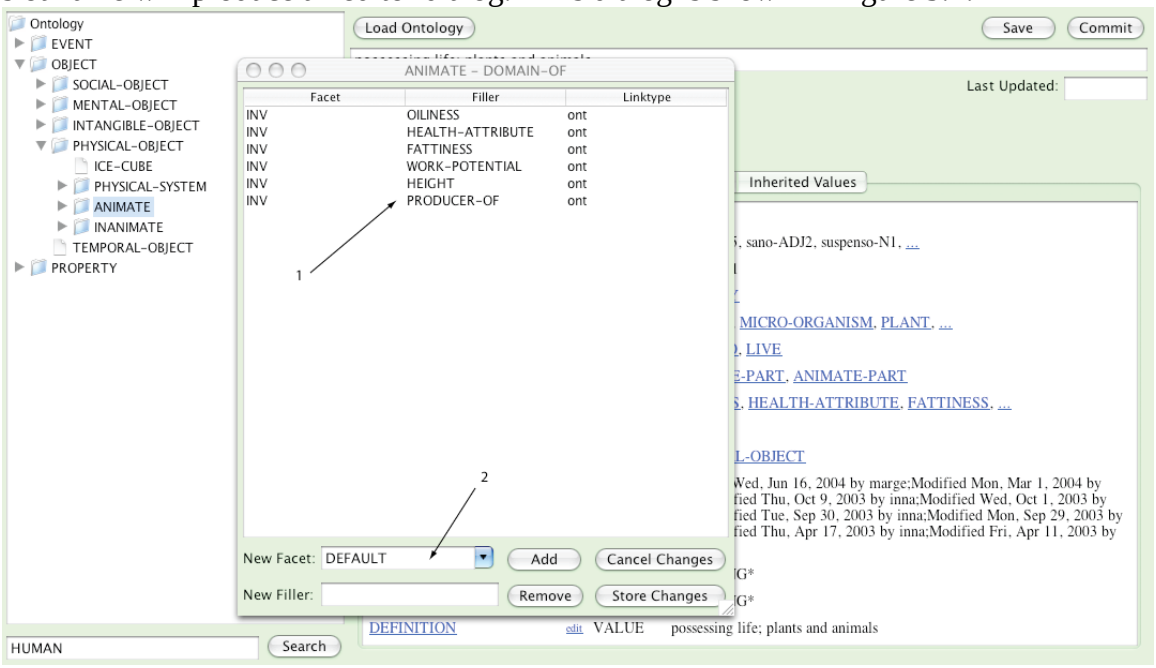

Figure 9.2: The Ontology Editor Dialog.

The ontology editor dialog displays all of the values related to a single slot in the current concept. Each of the values is directly modifiable in the table {Fig:9.2.1} by simply selecting the row and entering new data in the table cell. To add a new value, a drop list providing valid facet types is provided {Fig:9.2.2}. The user can also remove existing values entirely. When the user is done, the values are either stored or ignored depending on the user action, and the dialog is removed from the screen.

When the user wants to commit the modified ontological entries to the server, the user is presented with another dialog, containing each modified entry in table format. The commit dialog is shown in Figure 9.3.

The dialog for committing ontological entries displays a list of each modified entry, and allows the user to select which entries to commit {Fig:9.3.1}. The entry name is displayed, and a mouse-over produces the modified entry contents {Fig:9.3.2}. To the right of the entry name is a status column {Fig:9.3.3}, which changes during the course of committing the entry. The value starts as “not committed”, notifying the user that the entry is only stored locally. After selecting the commit button, each entry’s status is changed to either “validated” or “not valid”, reflecting the server’s response to validate the data found in the modified or new entry.

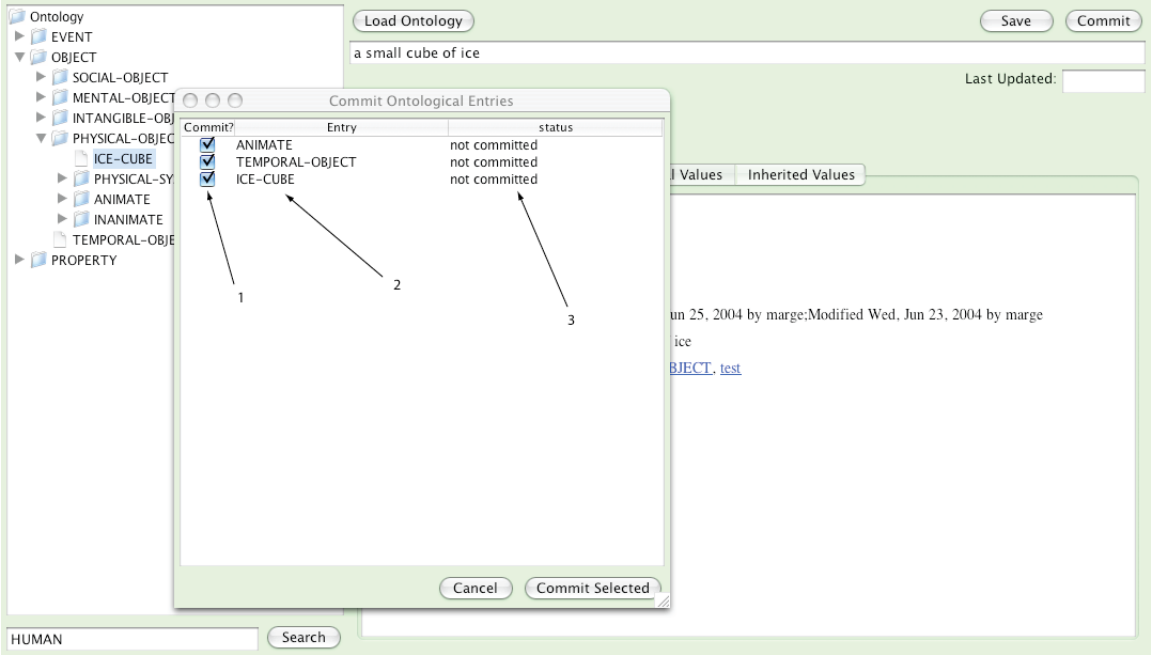

Figure 9.3: The Ontology Commit Dialog.

If the entry has been validated, the status will then change to "committed" or "not committed", reflecting the server's response to the timestamp provided by the client (this is to prevent the WAW scenario discussed in Chapter 7). In either the "not valid" or "not committed" scenario, the user can mouse over the status line to receive detailed information concerning the error.

10. The FR Interface

The fact repository interface's functionality is very closely matched to the functionality provided in the ontology interface. Even more than the ontology interface, however, the fact repository interface is truly designed as a browser. FR acquisition is left to one of two other possibilities: first, OntoSem does the majority of FR acquisition automatically.

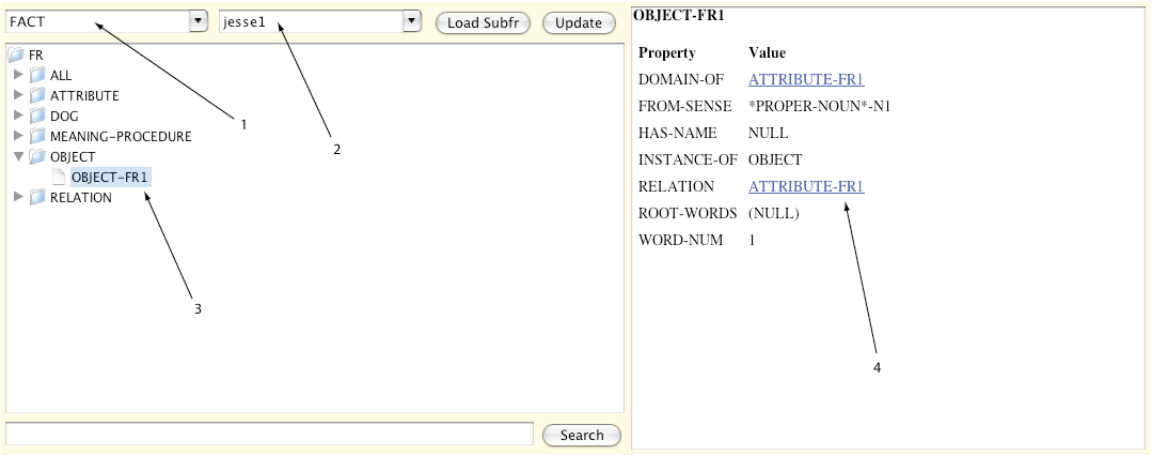

Figure 10.1: The FR Interface.

Second, specific research applications using OntoSem may require FR entries that are specially formatted (requiring certain properties to be present, possibly with limited ranges). To that end, the acquisition of application specific facts is better left to customized panels added to the DIIClient for the given application. One such panel is discussed in Chapter 12. Figure 10.1 shows the FR interface.

The interface provides the user with an intuitive view into the FR and subfr defined in the static knowledge. In the top left corner of the interface, a drop list of all known fact repositories is available {Fig:10.1.1}. After selecting a fact repository, the user is then presented with a list of all known subfrs inside the fact repository {Fig:10.1.2}. Once the data is loaded, a list of all concept instance types is provided, with each specific instance listed within {Fig:10.1.3}. The choice of display as a list, and not as a tree was made after careful examination of the type of data found in most fact repositories. Very few repositories had nearly enough instances of different concept types to justify constructing the view into a tree structure. Doing so would be not only computationally costly, but would also put more work on the user; a simple alphabetical list of concept types makes the work of examining various facts easier than forcing a user to root through a tree just to reach the category of desired instances. Like the ontology interface, any instances referenced in a fact repository entry are linked, to allow for easy browsing.

11. The OntoSem Interface

Of the four default interfaces in the DIIClient the OntoSem interface is the most feature rich. A successful interface into the analyzer would need to provide a way to send text to the analyzer, as well as to send modified output from any stage of the analysis. Each layer of output from the analyzer would need to be displayed in a user-friendly environment that also allowed for advanced editing. The solution was to create a separate graphical editor for each of the four stages (although the third and fourth stage are similar enough in output to allow for only one editor to be shared amongst them). The user is initially presented with the interface shown in Figure 11.1.

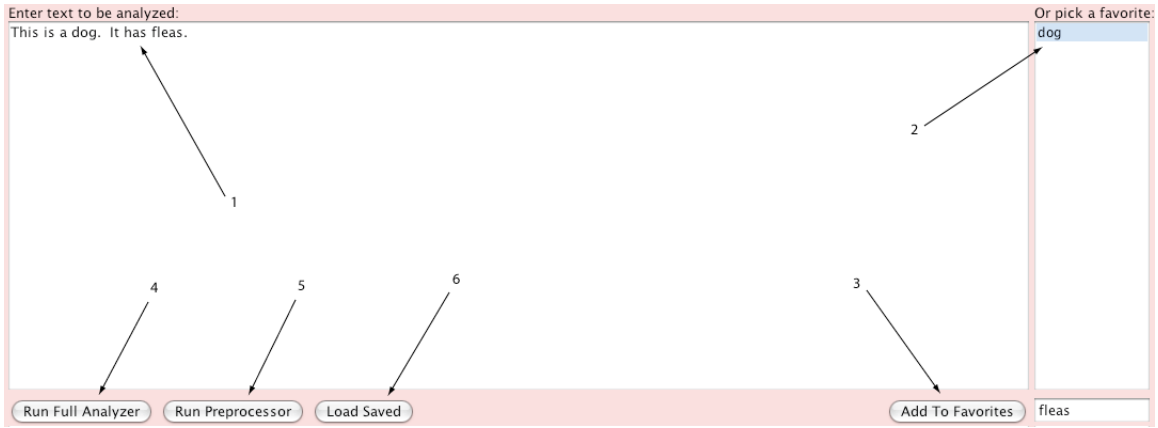

Figure 11.1: The OntoSem Input Interface.

To request analysis from OntoSem, a user must first provide a text in the large text box encompassing most of the interface {Fig:11.1.1}. The user is free to type any text

desired, copy and paste text into the box, or select a “favorite” text from the list to on the right {Fig:11.1.2}. Once the input text has been entered into the box, the user can choose to save the text as a favorite for the future {Fig:11.1.3}, or can begin to process the text in OntoSem. To process the text, the user can choose to run all four major processes on the text at once (requiring minimal interaction from the user) {Fig:11.1.4}, or to process only the first stage of the analysis, with the preprocessor {Fig:11.1.5}. If the user has already processed a text, and has saved the results for later, they can be retrieved by loading the saved file {Fig:11.1.6}. At this point, some analyzed results will now be available, and the interface changes as shown in Figure 11.2.

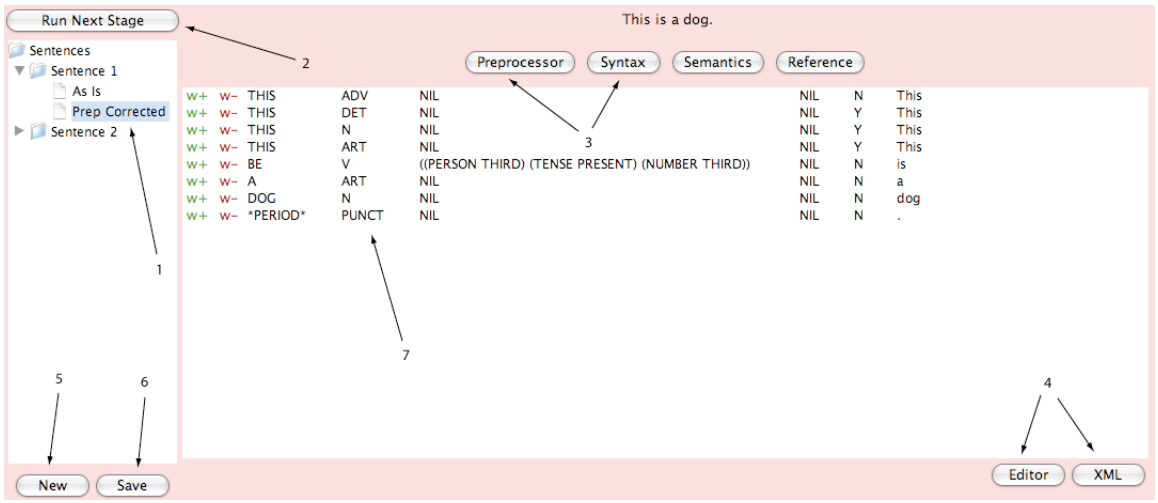

Figure 11.2: The OntoSem analyzed results and Preprocessor Editor.

To visually assist the user, the results returned from the analyzer are displayed in a shallow treelike structure. Each sentence in the text is independently displayed, and the results of each stage of manual correction of the analyzer’s results are displayed as leaves of the tree {Fig:11.2.1}. A user can select any leaf to have all available results of the sentence’s analysis at that level of modification displayed on the right side of the screen. To produce another level of results, the user can select to run the next stage of the analysis {Fig:11.2.2}. If the user wants to see the results of any of the processors at the current stage of analysis, a button for each available results is found at the top of the interface {Fig:11.2.3}. The results from the analyzer are originally available in XML file format (a structured, tag-based format similar in style to HTML). Each editor presents the processor’s output in a graphical format. To see or edit the original XML, the user can select to toggle the view in the lower right corner {Fig:11.2.4}. The user is also able to start a new text analysis (abandoning the old one) {Fig:11.2.5}, or to store the analysis for later display and modification {Fig:11.2.6}.

The preprocessor editor is a spruced up table, displaying the results of the preprocessor from the analyzer. Each possible part of speech matched to a word is displayed {Fig:11.2.7}. These values can be modified through editable drop down lists. Further, the user can add any missing analysis, and improper analysis can be manually removed as well. When the preprocessor modification is complete, and the user runs the analyzer on the modified output, the interface will display the syntax editor (Figure 11.3).

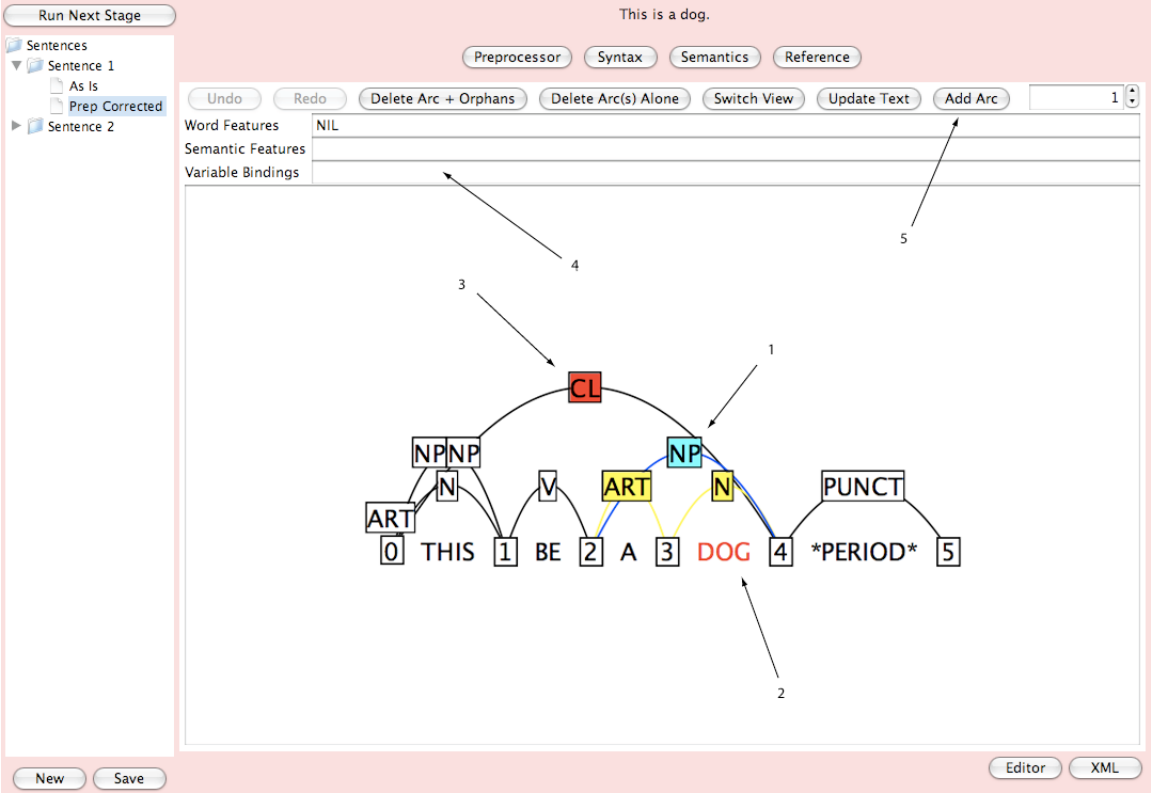

Figure 11.3: The Syntax Editor.

When the user decides to browse and edit the syntactic display from the analyzer, they are presented with the syntax editor. This editor displays each syntactic arc recognized by the analyzer, and allows the user to select any of them. A selected arc is highlighted in blue {Fig:11.3.1}, has its root word highlighted in the sentence {Fig:11.3.2}, its head phrase highlighted in red {Fig:11.3.3}, and displays any word features, semantic features, or variable bindings associated with arc above the graphical panel {Fig:11.3.4}. The user can modify any of these values, as well as remove and add arcs to the analysis {Fig:11.3.5}. After syntactic editing, the user is presented with the semantic/reference editor as shown in Figure 11.4.

The semantic/reference editor is designed to show the user relations between instances found inside the sentence. The user can navigate by displaying all relations related to an object or event, or display all relations of a certain type, found in the TMR {Fig:11.4.1}. Once selected, the attributes found in the TMR are displayed, and each relation matching the query is listed, showing the two associated instances, their root words (the words in the text they came from), and the relation type connecting them {Fig:11.4.2}. The user can change the relations by removing or adding new ones {Fig:11.4.3} to produce a modified TMR.

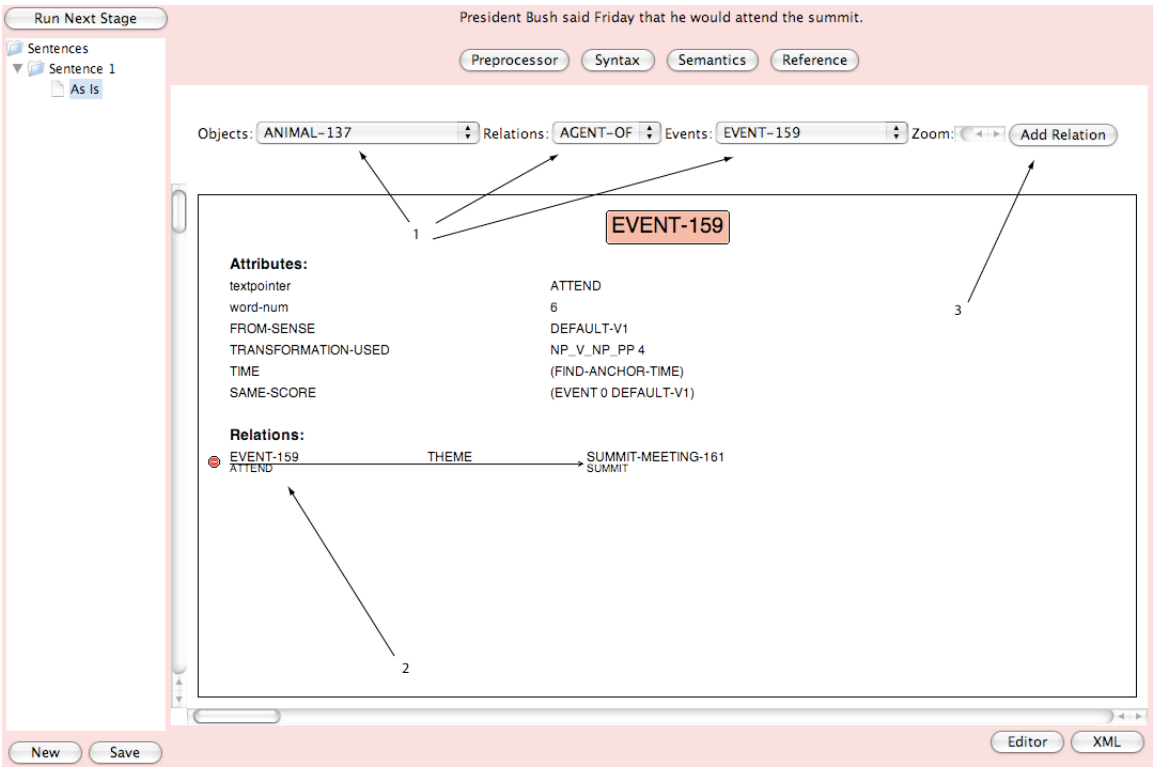

Figure 11.4: The Semantic/Reference Editor.

Once the user has modified the output of the analyzer, a golden TMR can be stored in the FR. An evaluation tool can now be used to test the output of the analyzer on the same target sentence against the golden TMR. This evaluation can help to determine the quality of automatically produced analyses. Intuitively, the evaluator can examine the corrected results against the automated results, and build a table of comparisons. The process of automatic text analysis evaluation is discussed further in [11].

The OntoSem interface is a highly interactive editing panel that allows the user to input text or modified results of any stage of analysis to the OntoSem analyzer, and to receive results from the server. The interface allows the user to produce golden TMRs as well as quickly and visually demonstrate the processing power of the OntoSem system.

12. Current Uses, Future Work, Conclusion

The DII system has seen usage even before a stable release of the DekadeAPI. A motivating force behind the system's creation was to allow researchers easy access to the OntoSem analyzer and supporting functions and knowledge. Two major research projects have used the DII architecture since its inception, with several smaller projects also contributing to DII's repertoire. The SemNews project is a web-based RSS news crawler, which feeds live headline news into OntoSem to create a semantic view of current events. The Medical Patient Creator is a custom FR building application intended to supplement an extension of OntoSem used for medical simulations.

12.1 SemNews

Developed by Akshay Java, et al. [9][10], the SemNews system scans user defined RSS news feeds for new headlines. Upon finding new data, the headline is sent through the DIIServer to OntoSem for analysis. A fact repository is then supplemented, and can be browsed through the SemNews interface. A screenshot of the SemNews interface fact-browsing page is shown in Figure 12.1.

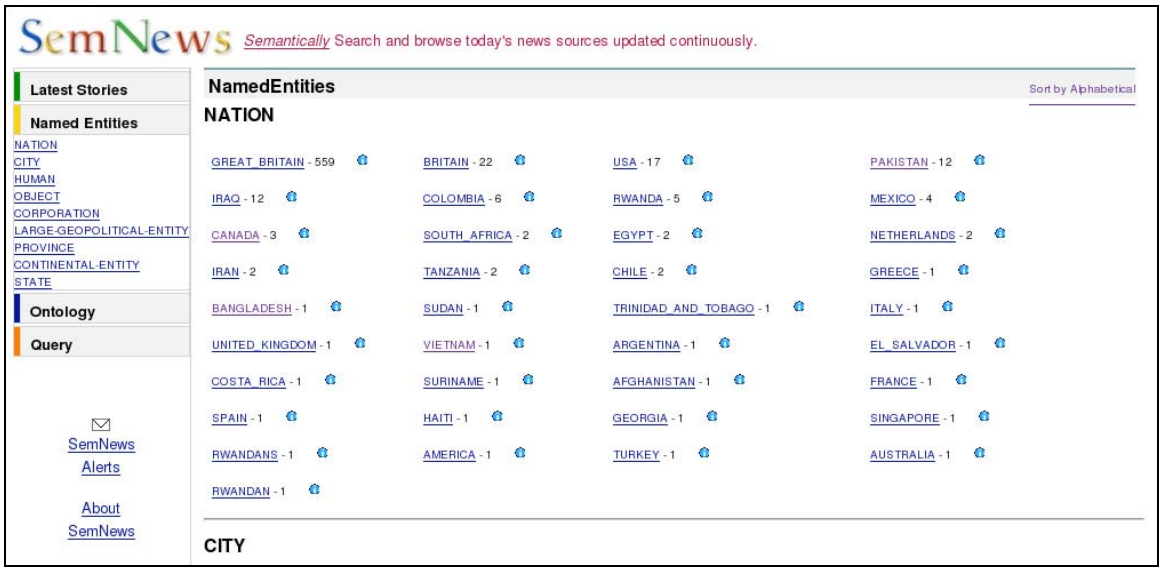

Figure 12.1: The SemNews Fact Repository Browser.

12.2 Medical Patient Creator

A prime example of custom fact repository creation templates in DII is the Medical Patient Creator (MPC). The interface was developed following the guidelines for all DIIClient interfaces, as well as a standalone application. The MPC allows the users to enter data in a form manner, as if filling out a patient history chart. The interface then outputs the data in both a format readable by the OntoSem medical simulator, as well as a FR instance. A screenshot of the MPC interface is shown in Figure 12.2.

12.3 Future Work

The open architecture and design of the DIIClient allows for a wide array of NLP related tools to be constructed. A notable feature that would vastly increase the expressive power of the OntoSem environment would be the addition of script editor to DII.

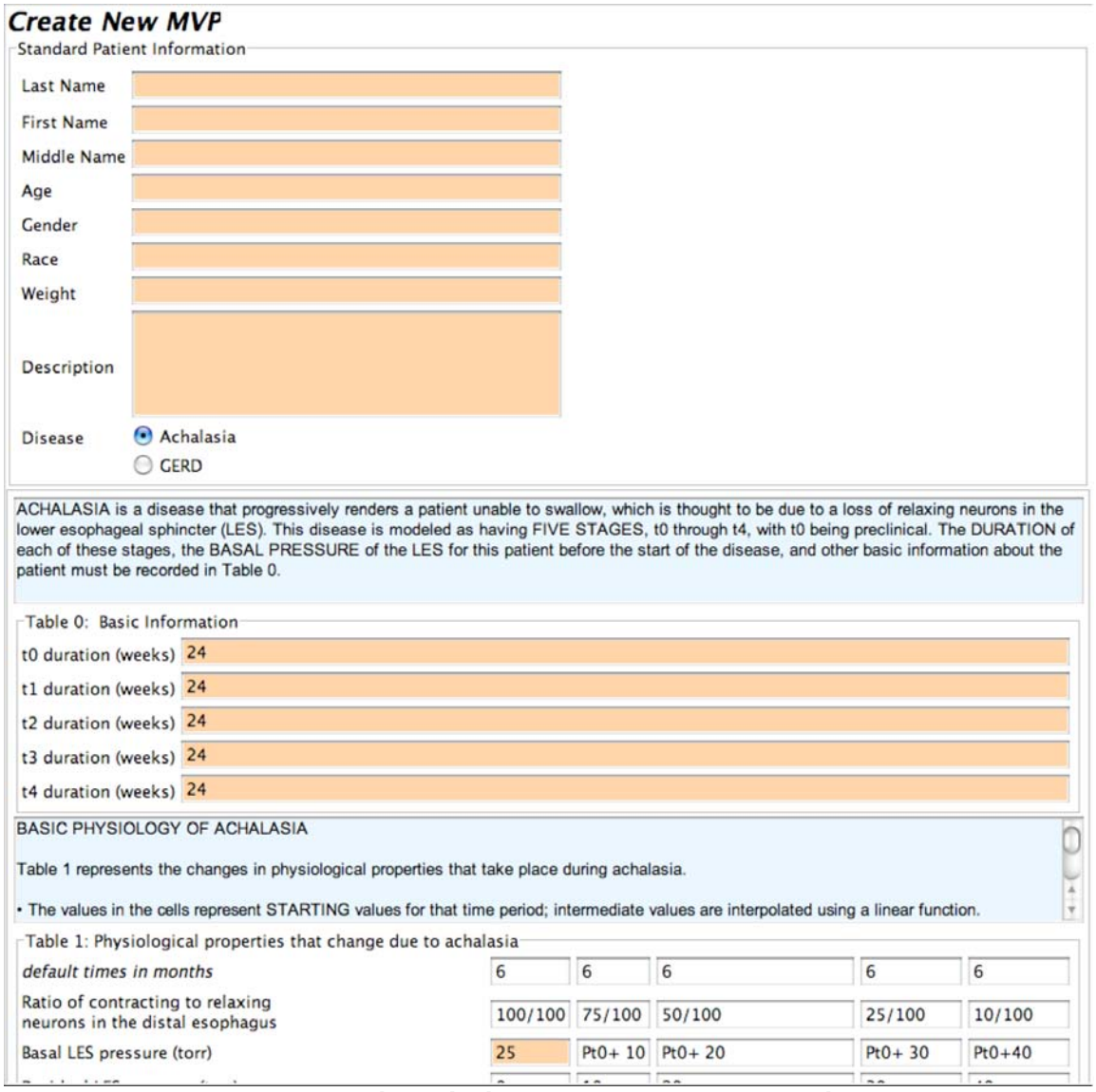

Figure 12.2: The Medical Patient Creator custom FR template.

A script, in this context, refers to series of ontological events with further defined (or constrained) property values, which define a logical event. An example would be “taking a flight”. At some grained level of expression, this could boil down to:

1. Purchasing a ticket
2. Packing luggage
3. Taxiing to the airport
4. Finding the terminal
5. Boarding the plane
6. Landing at the destination

Each of the above steps would be an instance of some *EVENT* concept, and there would be at least one, if not many actors, being instances of *HUMAN*. They would have

properties defined by constraints concerning the particular script. The DekadeAPI would allow for the development of an editor capable of constructing scripts of such detail.

Another major enhancement supporting the plug-in environment of the DIIClient interfaces will be the inclusion of an interface builder for users who are unfamiliar with Java and GUI programming. Building on the DekadeAPI, an additional library of GUI components with built-in functionality designed for use with OntoSem will be constructed. An example of a component could be an ontology tree; instead of having to define a JTree component in Swing, load the ontology, and map the structure into the Swing's native format, the component would do this for the user automatically.

For users wanting more flexibility than the interface builder provides, a default interface template will also be constructed to assist the user in rapidly creating a properly formed interface object to work with the DIIClient environment.

Additionally, the DekadeAPI will continue to undergo additions and enhancements to supplement the interface programmer's array of tools and to increase efficiency in data access and manipulation.

12.4 Conclusion

This paper has introduced the robust DII NLP development and demonstration system, which is comprised of a solid foundation library, as well as server side application and an open architecture client side GUI. The need for such an environment has been shown by rigorously examining the components of the OntoSem analyzer, and introducing the methods in which each of these components interacts with each other. Also introduced were the four fundamental interfaces provided by the DIIClient, and motivated the choice in designs behind each one from an end-user's perspective. Finally, the open design of the DII system allows for work to continue in the area of tools development, in order to aid various research projects that use the OntoSem analyzer or any of the static knowledge resources available to the system.

13. Acknowledgements

I would like to thank Sergei Nirenburg for guidance and assistance during this project. I would also like to thank Marjorie McShane, Stephen Beale, and Thomas O'Hara for acting as guinea pigs during the earlier stages of development. Finally I'd like to thank Donald Dimitroff for editing assistance on this report.

References

[1] The Evolution of Protégé: An Environment for Knowledge-Based Systems Devlopment. John H Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubezy, Henrik Eriksson, Natalya F. Noy, and Samson W. Tu. International Journal of Human-Computer Studies, 58. 2003.

[2] The Berkeley FrameNet Project. Collin F. Baker, Charles J. Fillmore, and John B. Lowe. Proceedings of the Thirty-Sixth Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics. 1998.

[3] The Annotation Graph Toolkit: Software Components for Building Linguistic Annotation Tools. Kazuaki Maeda, Steven Bird, Xiaoyi Ma, and Haejoong Lee. Proceedings of the first international conference on Human language technology research. 2001.

[4] TableTrans, MultiTrans, InterTrans, and TreeTrans: Diverse Tools Built on the Annotation Graph Toolkit. Steven Bird, Kazuaki Maeda, Xiaoyi Ma, Haejoong Lee, Beth Randall, and Salim Zayat. Proceedings of the Third International Conference on Language Resources and Evaluation, Paris: European Language Resources Association. 2002.

[5] Commonsense Reasoning in and over Natural Language. Hugo Liu and Push Singh. Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information & Engineering Systems. 2004.

[6] ConceptNet – a practical commonsense reasoning tool-kit. H. Liu and P. Singh. BT Technology Journal, To Appear. Volume 22.

[7] Ontological Semantics. Sergei Nirenburg and Victor Raskin. MIT Press. 2004.

[8] Open Mind Commonsense: Knowledge Acquisition From The General Public. P. Singh, T. Lin, E. T. Mueller, G. Lim, T. Perkins, and W. L. Zhu. Proceedings of the First International Converence on Ontologies, Database, and Applications of Semantics for Large Scale Information Systems. Lecture Notes in Computer Science No 2519, Heidelberg, Springer. 2002.

[9] Text understanding agents and the Semantic Web. Akshay Java, Tim Finin, and Sergei Nirenburg. Proceedings of the 29th Hawaii International Conference on System Sciences. 2006.

[10] Integrating Language Understanding Agents Into the Semantic Web. Akshay Java, Tim Finin, and Sergei Nirenburg. First International Symposium on Agents and the Semantic Web. AAAI Press. 2005.

[11] Evaluating the Performance of the OntoSem Semantic Analyzer. Sergei Nirenburg, Stephen Beale, and Marge McShane. Forthcoming in the Proceedings of the ACL Workshop on Text Meaning Representation. 2004.

[12] Semantically Rich Human-aided Machine Annotation. Marjorie McShane, Sergei Nirenburg, Stephen Beale, and Thomas O'Hara. Proceedings of the Workshop on Frontiers in Corpus Annotation II: Pie in the Sky, ACL-05. Pp. 68-75. 2005.