2010-01-01

Using Controlled Natural Language for World Knowledge Reasoning

Nelson Charles Dellis
University of Miami, nelson.dellis@gmail.com

Follow this and additional works at: https://scholarlyrepository.miami.edu/oa_theses

UNIVERSITY OF MIAMI

USING CONTROLLED NATURAL LANGUAGE FOR
WORLD KNOWLEDGE REASONING

By

Nelson Charles Dellis

A THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Master of Science

Coral Gables, Florida

June 2010

UNIVERSITY OF MIAMI

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

USING CONTROLLED NATURAL LANGUAGE FOR
WORLD KNOWLEDGE REASONING

Nelson Charles Dellis

Approved:

_____   _____

Geoff Sutcliffe, Ph.D.   Terri A. Scandura, Ph.D.
Professor of Computer Science   Dean of the Graduate School

_____   _____

Ubbo Visser, Ph.D.   Brad Cokelet, Ph.D
Professor of Computer Science   Professor of Philosophy

DELLIS, NELSON CHARLES   (M.S., Computer Science)
Using Controlled Natural Language for World Knowledge Reasoning   (June 2010)

Search engines are the most popular tools for finding answers to questions, but unfortunately they do not always provide complete direct answers. Answers often need to be extracted by the user, from the web pages returned by the search engine. This research addresses this problem, and shows how an automated theorem prover, combined with existing ontologies and the web, is able to reason about world knowledge and return direct answers to users' questions. The use of an automated theorem prover also allows more complex questions to be asked. Automated theorem provers that exhibit these capabilities are called World Knowledge Reasoning systems. This research discusses one such system, the CNL-WKR system. The CNL-WKR system uses the ACE controlled natural language as its user-input language. It then calls upon external sources on the web, as well as internal ontological sources, during the theorem proving process, in order to find answers. The system uses the automated theorem prover, SPASS-XDB. The result is a system that is capable of answering complex questions about the world.

Acknowledgements

I have a lot of people that I would like to thank dearly for their support and guidance. Firstly, I would like to thank my graduate advisor Professor Geoff Sutcliffe. Without his continued support over the past couple years, I would not be in the position to defend any thesis. His excitement for the research he does is contagious and has been incredibly inspiring and fun to be a part of. Secondly, I would like to thank the rest of my committee, Professor Ubbo Vissor and Professor Brad Cokelet. I would like to thank them for accepting a position on my committee and for taking the time to read and provide comments on my thesis, as well as hear my thesis defense. Next, I would like to thank the Department of Computer Science for giving me the opportunity to come back and finish my studies after a taking a year off. I would also like to thank the Department for making every effort possible to retain my assistantship with the department. Lastly, I would like to thank my close friends and family for helping me make it all the way through graduate school.

Contents

List of Figures

Chapter 1

Introduction

This chapter briefly presents some preliminary concepts of general knowledge reasoning, more specifically those of basic automated reasoning, and what it means to reason about general or world knowledge. This chapter also mentions Natural Language Processing (NLP) and how it is related to World Knowledge Reasoning (WKR). Section 1.1 defines WKR and its components, and discusses why such a type of reasoning is essential. Section 1.2 discusses the elements needed to construct a working WKR system. Section 1.3 discusses previous attempts, and progress towards, successful WKR systems. Section 1.4 gives an overview of the work done in this research. Section 1.5 outlines the contributions of this thesis. Finally, Section 1.6 outlines the structure of this thesis.

1.1 What is World Knowledge Reasoning?

To understand what WKR is, one must first understand the concept of world knowledge. There are alternative definitions of world knowledge, but for the purpose of this research, a specific one is given. *World knowledge* is naively, and circularly, knowledge about the world. It is knowledge that people commonly know or care to know in an everyday setting. World knowledge can be split into two types of knowledge, *factual knowledge* and *ontological knowledge*. Factual knowledge consists of basic facts about all the things in the world (e.g., the height of Mt. Everest, the number of hours in a day, the name of the author of a book, etc.). These facts are independent of one another, and each provides only one small portion of the world's available knowledge at a time. Ontological knowledge consists of rules and relationships between those facts, describing the structure of the world within which the factual knowledge resides. This research's definition of world knowledge allows for factual knowledge and relationships between two or more pieces of factual knowledge to be described in detail. For example, two separate pieces of factual knowledge might state the heights of two different mountains in the world, while a piece of ontological knowledge might describe that they are both a part of the same mountain range, or that if they have different heights then they are different mountains. It should be pointed out that in both of these types of world knowledge, there exists knowledge that is static and knowledge that is dynamic. Knowledge that remains unchanged for an infinite or nearly infinite amount of time is considered static. This pertains to things like 'circles are round' or 'green is a result of mixing blue and yellow.' Knowledge that has

the potential to change is considered dynamic. Dynamic knowledge are things like 'the weather of a city X is Y' or 'the world record for a sport X is Y.'

One can imagine the plentitude of factual knowledge and ontological knowledge that exists, but it might be hard to imagine the vastness of new knowledge that could be inferred by reasoning over those elements of world knowledge. WKR is reasoning done specifically on world knowledge, with the goal of inferring new and useful knowledge. WKR also aims to answer specific questions about world knowledge. One could draw parallels between WKR and how the human brain learns and answers questions that are posed to it. Using a finite sized bank of world knowledge, the brain can infer an almost limitless amount of new knowledge.

The usefulness of having a system that can perform WKR and deliver world knowledge efficiently to a user is obvious. What a WKR system creates is a single tool that can provide answers to all queries given by a user without the user having to look through or use numerous sources to arrive at an answer. In an era where world knowledge is essentially at the fingertips of anyone who has access to a search engine, WKR systems provide an alternative way to access knowledge much more easily and quicker than current methods. Currently, the most common method of retrieving answers to world knowledge queries are via search engines. Unfortunately, search engines do not provide direct answers or sometimes even any answer at all. The need for a system that can retrieve a piece of world knowledge without the user having to dig deep for it through a sea of URLs is the future.

WKR systems are useful in a number of domains. The primary domain is that of the general population, using the system as an addition to or even a replacement for

search engines. The secondary domain would be for businesses or organizations that require a large amount of static and/or dynamic knowledge about the world in order to operate efficiently and/or successfully.

1.2   What is needed for a WKR System?

A WKR system is made up of three parts: A front-end user interface, a source of world knowledge, and a reasoning engine. Each of these parts is discussed in detail in Chapter 2.

One of the more important parts of a WKR system is its user interface; the piece of software that interacts directly with the user. It is important because if a WKR system is going to be successful and considered a useful tool, it must, above all, be easy to use and have a user-friendly interface. Having a system that requires a lot of time to learn and understand, negates the benefits it provides. There are much faster ways to retrieve world knowledge that don't involve having to learn a specific query language or the intricacies of a system. One of the main purposes of a WKR system is to make knowledge retrieval quicker and more powerful than current systems. The ideal user interface is one where a user can enter pure English (or at least something that is intuitively close to English) into an input field and, without much configuring, can submit it to the WKR system and await the results. The results must also be presented back to the user in an easy-to-understand way.

Next, a source of world knowledge is required. The WKR system needs some place that it can retrieve world knowledge from, in order to be functional. Here is where

different systems have taken different approaches. Some systems have constructed their own knowledge databases while other systems search through a single or multiple sources to find the same knowledge. The problem with this part of the WKR system is that for it to be a perfect system, it must have knowledge about every domain in the world. This is nearly impossible to achieve, so the next best thing is having as close to that as possible. The more world knowledge a WKR system can retrieve, the better it can answer all queries about things and concepts of things in the world. Having more and more accessible world knowledge brings all sorts of difficulties with it though. For one, the larger a database of world knowledge becomes, the harder it is to search for a specific piece of knowledge and subsequently, the slower the system becomes.

Finally, an Automated Theorem Prover (ATP) or, reasoner, is needed to complete the system. One of the reasons why having a reasoner is so important is that it reduces the task described in the previous paragraph - the necessity of building a extremely large domain of world knowledge. The reasoner can use a smaller set of world knowledge instead, and infer a new and possibly more complex set of knowledge. This shrinks the difficulty posed previously as it no longer is necessary to capture the entire world in a single database. Instead, more importance can be placed on formalizing only small pieces of world knowledge, while relying on the reasoner to produce the larger or more complex pieces.

The reasoner is also used as the tool that finds the answers to a queries given by a user. The reasoner tries to use the world knowledge and inferred knowledge that is available to it to generate a proof of a conjecture (i.e., an answer to a query). Since

the user would be using the WKR system to answer any question he/she can think of, sometimes the query might be a simple piece of factual or ontological knowledge, in which case it is simply retrieved and provided back to the user. Other times, the user might be asking a query whose answer might not be readily available in the system's sources so it might need to perform numerous reasoning steps in order to provide a result.
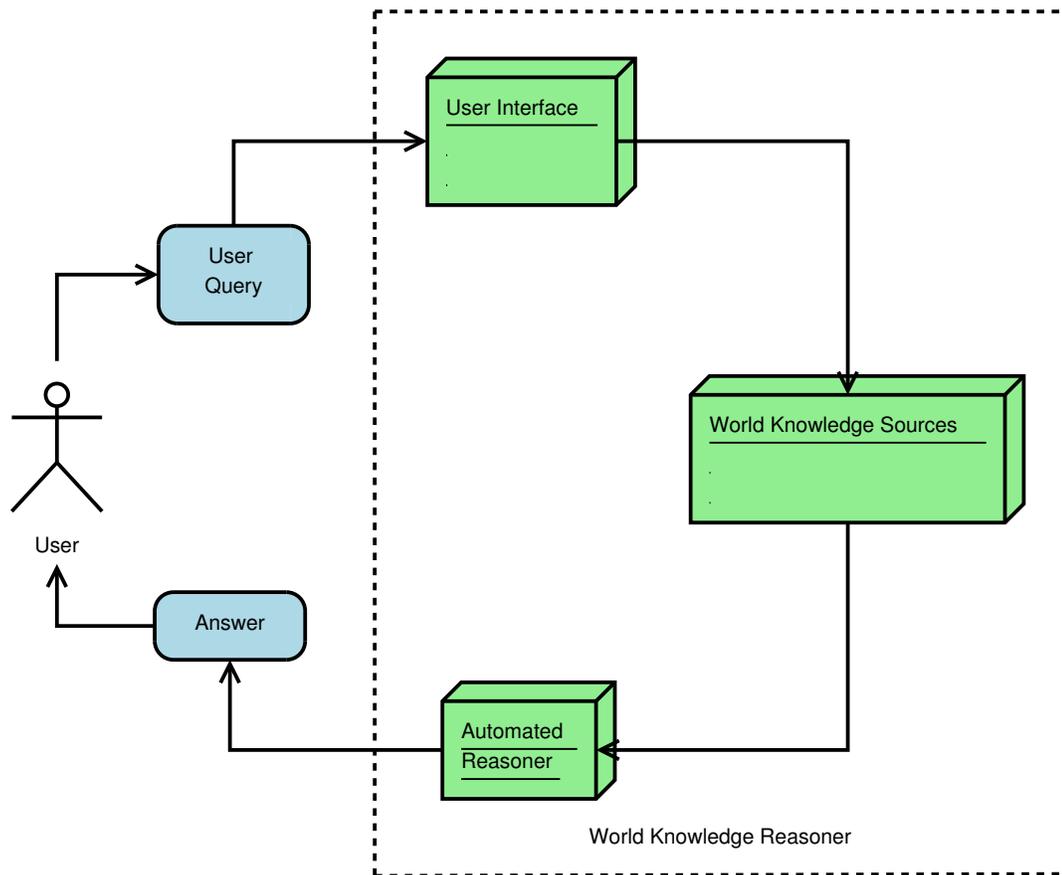
Figure 1.1: Basic structure of a WKR system

With these three structural parts, as shown in Figure 1.1, one can have a complete end-to-end system that is capable of answering world knowledge queries. The ideal situation would be to have a user-interface that uses a highly expressive query

language and a reasoner that is extremely robust (robust in the sense that a theorem prover can use a highly expressive language while maintaining . Unfortunately, a system cannot have both; the more expressive the language used, the harder it is to reason over. Making a system that allows highly complex queries allows for more complex results, but finding the answer to those queries might be more difficult, if not, impossible. Allowing for a simpler query language gives the system a much higher rate of success, but limits the expressivity of the system's results.

1.3 Overview of Previous WKR Systems

WKR systems have been studied and implemented before, and there are even a few that are currently operational and that see a fair amount of usage by the general public daily. Even though WKR systems exist, all of them have their limits and shortcomings. The current systems that can do WKR successfully, can reason over only small domains of world knowledge. Some choose smaller domains purposefully so that they can provide results over a smaller amount of world knowledge, while others choose to capture larger domains of world knowledge but are not as successful in providing results. As stated earlier, being able to represent the entirety of world knowledge in a system is extremely difficult to do, so it should be noted that systems that do WKR are not perfect.

One can first look at the simplest of systems, those being search engines. While they perform little to no reasoning at all, they still can be seen as a source of world knowledge retrieval. They are probably the most widely used tool to retrieve knowl-

edge and also the most attractive. Search engines, like Google[1], are fast and effective if the query is relatively common or simple. A search engine is not as effective when queries becomes more complex. For a query like 'name a city in France where the weather is currently sunny,' a search engine might return a set of URLs that contain the current weather data for different areas of France, but the user would still have to find the answer within that data. These kind of queries can often take a good amount of research on the user's part to find a suitable result. In recent years though, search engines have become smarter and more efficient. Newer search engines, like Bing[2], claim to be smarter and claim to have the ability to think and do basic reasoning.

An improvement from search engines are systems that use a limited knowledge-base or ontology, and that use reasoning to answer queries about the available knowledge. This is more like the idea of a full-fledged WKR. Examples of these systems are Cyc [1], True Knowledge[3], or Wolfram Alpha[4]. The problem with these systems is that they do not embody the entire corpus of human knowledge. They have attempted to formalize as much as possible about the world into a single database, but since each of these databases are constantly being updated and new knowledge is constantly being added to them, they keep increasing in size. This increase in size increases the search space for the system when it attempts to reason. Some say that the dream of formalizing all world knowledge is ridiculous and impossible, which is why even though these systems have seen some impressive results, their future is not very bright [2]. They can each answer a large amount of queries and are very

---

[1] http://www.google.com
[2] http://www.bing.com
[3] http://www.trueknowledge.com
[4] http://www.wolframalpha.com

successful, but have very high ambitions.

The alternative to systems that use a single massive knowledge database are systems that have none and instead, rely on numerous sources for world knowledge including the web and readily available ontologies, namely WKR systems. The web is a great source of information because it is constantly growing and being updated. There is no need to have large amounts of knowledge handwritten or curated because it is already available and ready to be used on the web. For example, Amazon[5] offers a web service which provides structured product information data to users.

1.4   Overview of this Work

This result of this work is the Controlled Natural Language World Knowledge Reasoning (CNL-WKR[6]) system.

Because human interaction with the system is a key part of its usefulness, the Attempto Controlled English (ACE) language was chosen as the query language (for several reasons, which will be discussed in Chapter 2). The advantages of this are that it is an easy language for users to learn, and it is a highly expressive natural language that doesn't have any ambiguity issues. Once the query is submitted by the user to the CNL-WKR system, the ACE query is then translated into first-order logic, the formalized language required for the reasoner to reason. The Nelson's ACE (NACE) translator was developed for this step and is discussed in depth in Chapter 3.

---

[5] http://www.amazon.com
[6] Pronounced 'Colonel Wacker.'

Once the translation has occurred, the formalized query is then fed into the theorem prover, SPASS-XDB, in the form of a conjecture. The theorem prover then fetches the necessary pieces of world knowledge as they are needed, in the form of axioms, from the available sources. The CNL-WKR system uses two types of sources, external and internal. The internal source is the Suggested Upper Merged Ontology (SUMO), which is an ontology that contains millions of world knowledge facts and relationships between those facts [3]. Contrastingly, the external sources come from a variety of places; some come from databases while others come from the web. For SPASS-XDB to be able to fetch world knowledge from the external sources, mediators were implemented. The mediators serve as interfaces between SPASS-XDB and the external sources. Each mediator is designed to connect to a specific domain of knowledge. These domains range from geographic data to up-to-date currency conversion rates. The details of SPASS-XDB are discussed in Chapter 3. Once all the additional axioms necessary to prove the conjecture (i.e., to answer the query) are amassed, the conjecture and the axioms are then fed to SPASS-XDB, which attempts to find a proof. The results are then returned to the user.

The CNL-WKR system can achieve powerful results, and has the potential to achieve more in the future. The number of mediators that are available for SPASS-XDB dictates the vastness of knowledge that it can reason over. Since mediators are relatively quick and easy to create, this is something that is not a concern for the moment. The emphasis is to create a WKR system that works from end-to-end. The ability for SPASS-XDB to prove more theorems partially relies on the growth of the mediator library. Once the mediator library has been expanded, the CNL-WKR

system will be able to increase its capabilities even more.

The limits of the the CNL-WKR system come hand-in-hand with the language it uses to formalize the user's queries: ACE. ACE is very expressive but, like any controlled language, it has constraints that don't allow certain phrases to be expressed. This means that queries must be formulated carefully. ACE is an unambiguous language and all ACE sentences or queries have one and only one interpretation. If a different interpretation is desired, then the ACE query must be written with different wording. One thing to be aware of is that when translating an ACE query to first-order logic, meaning can sometimes be lost in the process. This is one of the many difficulties of translating from a natural language to a logic language. Logic is rigid, and the operators it uses do not always necessarily convey the exact meaning that was originally desired.

The limits of the system also lie within the theorem prover. SPASS-XDB is a good theorem prover, but is not capable of proving everything. What is attempted in this research is a WKR system that is easy to use, that can provide useful information to the user, and that works fluidly from end to end.

1.5   Contribution of this Thesis

In this thesis, the complexities of creating an end-to-end WKR system are uncovered and discussed. The aim of this project is to create an end-to-end product where a user can ask a query in an English-like language on one end, and receive a specific answer expressed in the same English-like language on the other end. The strength of the

reasoner used in this process and how it can be improved is not explored here for it is not the focus of this thesis. The main focus points are English-to-logic translation, the difficulties with aligning the translated queries with the external factual knowledge sources and the ontological knowledge sources provided by SUMO. On a larger scale, this thesis helps provide a positive direction for the future of WKR systems. This thesis provides a directed approach at doing this with a good amount of success and also points out possible future difficulties that may arise.

1.6   Thesis Structure

The organization of the rest of this thesis is as follows: Chapter 2 reviews previous attempts at translating natural language into logic, a description of past and current natural language interfaces, and the improvements in WKR over the years. Chapter 3 discusses the design and implementation of the CNL-WKR system, including the steps needed to create this end-to-end piece of software. Chapter 4 discusses the testing of query translations and query reasoning, and describes the results and issues found with both. Finally, the thesis is concluded in Chapter 5 by discussing the limitations of the work and possible future work.

Chapter 2

Literature Survey

This chapter presents a review of how natural language has been approached in order to translate it into a logic form, the history and use of natural language interfaces, and finally how WKR has been approached and how it works. Section 2.1 gives an introduction to logic and explains why it is useful for this research. Section 2.2 discusses how attempts at translating natural language into logic have succeeded or fallen short. More specifically, the section focuses on how certain constrained subsets of English have been used to facilitate translation between English and logic. First, the section focuses on what Controlled Natural Languages (CNL) are available and why one would choose these languages as a prime candidate for translation. Two of them are of interest for this work, namely Attempto Controlled English (ACE) and Controlled English Logic Translation (CELT). Next, Section 2.3 discusses the history of natural language interfaces and how they have progressed, with emphasis on interfaces that deal with controlled languages. Section 2.4 reviews the history of WKR and why it has become so important in recent years. Finally, Section 2.5

provides a brief summary of the chapter.

2.1 Logic

Logic is the formalized representation of a statement or set of statements in a natural language by use of different combinations of symbols, operators, quantifiers, and variables. It is also the formalized language that SPASS-XDB uses.

The original motive for using logic was to study objective laws of logical consequence [4]. There are other important reasons for using logic instead of natural language. For one, it provides a high-level language in which knowledge can be expressed in a transparent way. Logic also has well-understood formal semantics, which assigns an unambiguous meaning to logical statements. Another reason logic is used is because there is a precise notion of logical consequence, which can determine whether a statement follows deductively from a set of other statements (the premises) [5]. Depending on the expressivity of a form of logic, sound and complete proof systems exist. Because these proof systems exist, one can use these proofs to provide explanations to answers. In relation to this research, being able to provide a proof for a given query not only yields an answer, but also provides a logical deduction process for why the answer is so.

Propositional logic, which is the lowest order logic that exists, is the simplest way to express natural language sentences in logic. By using connectives such as conjunction, disjunction, and/or conditional operators to connect statements, one can create the desired representation of a natural language in logic [6]. Each of the

statements is either true or false, and combined with the logical connectives, can produce different logical outcomes. Propositional logic looks at the truth values of logical formulae which are built from a combination of simpler statements that are connected by logical connectives. While this type of logic is easy to translate and easy to test for logical consequence, the expressivity the logic offers is weak and lacking.

First-order logic, which is the next step up from propositional logic, allows for quantifiers. Quantifiers allow for things to be talked about as a single instance (*exists*) or as universal generalizations (*for all*).

Although first-order logic is more expressive, as a logic language gets richer, it loses decidability [7]. In other words, it becomes harder to check for logical consequence. It can be shown if a formula is a logical consequence of a set of axioms, but it isn't necessarily possible to show that a formula is not a logical consequence of a set of axioms. First-order logic is also harder to test for logical consequence. This is the trend of higher-order logics, that as they increase in expressivity, the ease with which the language can be analyzed diminishes.

Translating from natural language to first-order logic is not a trivial matter, but can be reduced into a relatively mechanical procedure. In first-order logic, there are three types of information: *variables*, *functors*, and *predicates* [7]. Functors are terms used to describe objects in a domain of interest. For example:

Example 2.1 *president_of(america)*

In Example 2.1 the term *president_of* is a functor and contains another term, *america*. *America* is placed inside *president_of* because the president of America is an

object itself. Functors differ from predicates because predicates indicate relationships between two or more terms or objects in a domain of interest while functors simply describe one object at a time. In contrast to Example 2.1 a predicate could be:

Example 2.2 *president_of(obama, america)*

Example 2.2 denotes that the Obama is the president of America. The *president_of* predicate links the object Obama with the object America, describing a relation between them that says one is the president of the other.

The arguments of a functor or predicate do not have to be other defined terms, they can also be undefined variables [7]. These variables can then be either existentially or universally quantified. For example:

Example 2.3 $\exists X$: *president_of(X, america)*

or

Example 2.4 $\forall Y \exists X$: *president_of(X,Y)*

Example 2.3 denotes that there exists an X such that X is the president of America, or that there is someone who is the president of America. Example 2.4 denotes that for every Y, there exists an X such that X is the president of each Y, or that for all countries there is someone whose is president. Both examples demonstrate the expressiveness of first-order logic over predicate logic. Because SPASS-XDB uses an expressive logic language, it is capable of proving some complex problems.

Not only does SPASS-XDB require first-order logic as its input, it specifically requires TPTP [8] first-order logic. TPTP first-order logic is a format standard for

the first-order logic language and was developed by Geoff Sutcliffe and Christian Suttner. The TPTP format encloses a first-order logic formula within the following structure and also includes the name of the formula and the type of formula (e.g., `axiom` or `conjecture`):

```
fof(Name,Type,Formula)
```

TPTP format also uses a question mark (`?`) to represent existential quantifiers, and an exclamation mark (`!`) to represent universal quantifiers. Conjunction is represented by an ampersand (`&`), and disjunction by a vertical bar (`|`). This standard is used throughout this work and shown in the following example (taken from the Chapter 4 testing results):

Example 2.5

```
fof(name1,conjecture,(
    ! [A,B,C] :
      ( ( man(A)
        & man(B)
        & man(C) )
     => ( A = B
        | A = C
        | B = C ) ) )).
```

2.2   Natural Language to Logic Translation

Converting natural language to logic has long been of importance. This research is just one example that shows the importance of being able to perform this conversion. To be able to use SPASS-XDB, the ATP that is a part of the CNL-WKR system, the

user's English query must be converted to a first-order logic form so that reasoning can take place. ATPs generate proofs and answer queries, as long as the query that is being fed to it is in the form of a formalized conjecture.

Conversion between natural language and logic is not entirely straightforward. The problem with translating English into logic has been that English, as do many other languages, has too many ambiguities and idioms. To successfully interpret a language, one must be able to account for all possibly intended meanings and usages of parts of speech. To do this, one must add order and rules to a language in a strict manner. The language must be reduced down to something that has no exceptions, no alternate meanings, a restricted grammar, and that is not nearly as complex.

One approach that has been made in reducing the complexity of the English language has been by allowing only very specific sentence structures. Forcing English to be constrained allows for a system to be interpreted and understood without confusion. While there are a number of controlled English languages available, such as ACE[1], CELT [9], Rabbit [10], and PENG [10], the two that are of interest to this research are ACE and CELT. ACE is of interest because of its simplicity and expressiveness. CELT is of interest as well because it is linked to SUMO and is almost equally as expressive as ACE.

2.2.1   What is a Controlled Natural Language?

A Controlled Natural Language (CNL) is a language where the use of words, grammar, and/or meanings of words and phrases is subject to numerous constraints. For the

---

[1]`http://attempto.ifi.uzh.ch/site/`

purpose of this thesis, English CNLs will be discussed, but it should be noted that CNLs also exists in other languages, such as French [9], German [9], and Chinese [9]. CNLs are typically used to create clear and universally understood language. This is achieved by greatly reducing the complexity of sentence structures and standardizing the rules of their interpretation. Using CNLs is useful in drafting documents that need to be interpreted in a specific way, and that cannot afford to be written in a potentially unclear manner, such as product manuals or legislative documents [9]. The ambiguities that naturally exist in all languages are many, and that is why CNLs are necessary.

One of the biggest difficulties with processing a language is understanding its semantics. For example, consider the following sentence:

Example 2.6 *A woman hits a man with a bat.*

This sentence seems clear enough, but it may be interpreted in two different ways. It is not clear whether the woman is hitting a man by use of a bat or if she is hitting a man who is carrying a bat. If an ATP were to use this sentence in its translated logical form, the translated logic might not represent the originally intended semantics, and therefore the system might not be able to prove the desired result. It would be here that the rules of a CNL would force the above sentence to be interpreted one way and not the other.

Other advantages of using CNLs are that the information expressed in the language can easily be retrieved and re-used with the same exact intention and meaning at a later date. CNLs allow for repeated consistent translations. If something is writ-

ten in a CNL, then when another person needs to go back and read it, it is guaranteed that the reader will understand and be able to interpret the text in the same original way.

The advantages of using CNLs are great, but there are also some disadvantages. Many who use controlled languages at first tend to have difficulties learning them, and can often find it frustrating. Not only do they have to learn the language, they must also learn how to use the tools that are accompanied with them [11]. CNLs can also lose a lot of the original languages' nuances and aesthetics, thereby making it a language that is more rigid and sometimes clumsy in feel.

2.2.2 Attempto Controlled English

ACE is a subset of English, and a controlled version of the language. It was developed by Norbert Fuchs at the University of Zurich, Switzerland, in 1995 [12]. ACE has a grammar that constrains the meanings of all its acceptable sentences. In other words, all acceptable ACE sentences have exactly one interpretation [12]. ACE was designed "as a specification language that combines the familiarity of natural language with the rigor of formal specification languages" [13].

The ACE language is made up of two types of words, *function words* and *content words*. Function words are key single words or groups of words that are pre-built into the language. When used, they force meaning onto the entire sentence, like negation, quantification, implication, etc. (e.g., *not*, *there exists…*,*it is possible that…*). Content words are the nouns, verbs, prepositions, adverbs, and adjectives that make up the

sentences around the function words[2]. The Attempto Parsing Engine (APE)[3] is a tool that parses and tokenizes ACE sentences to check if they are valid. APE contains a basic lexicon of words, which includes information about parts of speech, tenses, and plurals of words. Words that are not recognized by the lexicon can by tagged by the user with a prefix to indicate a property of the word, such as its intended part of speech[4]. For example:

Example 2.7 *The man has a n:birthdate.*

In Example 2.7, the lexicon does not recognize the term *birthdate*, so the user must include a prefix that indicates that it is a noun.

Among the possible sentences that are accepted by the ACE language, one group of them is simple sentences involving subject-verbs. Direct or indirect objects, transitive or ditransitive verbs, adjectives, adverbs, or preposition can be added to these sentences in order to modify them even further. For example:

Example 2.8 *The man drinks.*

Example 2.8 demonstrates the simplest of statements that include one object (i.e., *man*) and one verb (i.e., *drinks*). Sentences can also be created by including function words that indicate universal or existential quantification. For example:

Example 2.9 *There exists a dog.*

Example 2.10 *All cows eat grass.*

---

[2]http://attempto.ifi.uzh.ch/site/docs/ace_constructionrules.html
[3]http://attempto.ifi.uzh.ch/site/tools/
[4]http://attempto.ifi.uzh.ch/site/docs/ace_nutshell.html

Similarly, simple sentences can be conjoined and disjoined simply by adding 'and' or 'or'. The ACE language can also distinguish between using 'and' as a conjunction and using it as part of a plural object (i.e., *the cat and dog...*). Also, similar to logic, 'and' binds stronger than 'or', but can be over-ridden by use of a comma[5]. The following are other valid ACE sentences:

Example 2.11 *The man drinks a beer and the child cries.*

Example 2.12 *The man drinks a beer and a soda.*

Example 2.13 *There is some man who wins or there is some woman who wins.*

Example 2.14 *A man inserts a VisaCard or inserts a MasterCard, and inserts a code.*

It should be noted that ACE does not deal with the temporality of statements. In Example 2.11, it could be assumed that the child cries as a result of the man drinking a beer, suggesting that the beer was drunk first and then the child cried. ACE does not make this distinction. An extension of this issue is made apparent by the fact that ACE cannot, as of yet, accept sentences in tenses other than the present.

In addition to conjunction and disjunction, ACE also can express other common expressive features such as conditionality, modality, negation, interrogative questions, and imperative statements[6]. ACE can also deal with plural nouns. Observe the following valid sentences:

---

[5]`http://attempto.ifi.uzh.ch/site/docs/ace_nutshell.html`
[6]`http://attempto.ifi.uzh.ch/site/docs/ace_nutshell.html`

Example 2.15 *If the pencil falls then it hits the ground.*

Example 2.16 *It is possible that the pencil hits the ground.*

Example 2.17 *The two pencils do not hit the ground.*

Example 2.18 *Does the pencil hit the ground?*

Example 2.19 *John, drop the pencil!*

Example 2.15 displays conditionality, Example 2.16 displays modality, Example 2.17 displays negation, Example 4.14 displays a question, and Example 2.19 displays an imperative statement.

One of the best features of ACE is its ability to avoid issues of ambiguity. To avoid ambiguity issues, ACE uses what is called the 'principle of surface order' [12]. This principle makes the scope of a quantifier easy to predict based on its position in the sentence. The scope of a quantifier begins at the textual position of the noun phrase being quantified and carries all the way to the end of the sentence. Example 2.20 shows a commonly used, ambiguous sentence:

Example 2.20 *Every man loves a woman.*

In ACE, Example 2.20 is interpreted unambiguously as the following logical form:

$$\forall X : (man(X) \implies \exists Y : (woman(Y) \ \& \ loves(X,Y)))$$

But, there might be an alternate interpretation desired, such as:

$$\exists Y : (woman(Y) \ \& \ \forall X : (man(X) \implies loves(X,Y)))$$

To achieve the latter interpretation, ACE includes two extra function words that moves the quantifier to the front of the sentence, in order to give it a wider scope. These function words are *there is/there are* and *for every/for each* [12]. In ACE, the second interpretation can be expressed as:

Example 2.21 *There is a woman such that every man loves her.*

or

Example 2.22 *There is a woman that every man loves.*

Even though the ACE language avoids ambiguities and will reject sentences that are not written with the correct ACE grammar, it can accept some ambiguously written statements and transform them into a form that makes them clear. These ambiguities are dealt with in a few ways. First, if a user writes an ambiguous sentence, the sentence is rewritten by APE in an unambiguous way, made up of smaller unambiguous sentence constructs. Secondly, for the few ambiguous sentences that ACE accepts, it applies some basic interpretation rules. These rules are consistent and will produce the same disambiguation of a sentence each time[7]. If the user is not satisfied with ACE's disambiguation, the user must rewrite the sentence. For example, ACE converts the following ambiguous statement:

Example 2.23 *A man eats a chicken that is healthy and drinks a soda.*

into:

---
[7]`http://attempto.ifi.uzh.ch/site/docs/ace_nutshell.html`

Example 2.24 *There is a man. There is a chicken. The man eats the chicken. The chicken is healthy. The man drinks a soda.*

ACE understands that the sentence is supposed to be saying that a man eats a chicken that is healthy and that the man drinks a soda, not that the man eats a chicken that is healthy and drinks a soda. If the latter was meant, then the ACE sentence should be explicit:

Example 2.25 *A man eats a chicken that is healthy and the chicken drinks a soda.*

The user can also write the second statement with the word "and" in place of periods. ACE's interpretation rules are meant to deal with sentences that are not so easily converted into simpler constructs such as above. Knowing all of the specific interpretation rules is not important since all sentences can be rewritten if a problem arises.

ACE is a very desirable language for the subject of this research. It is a language that is easy and intuitive to learn. When a sentence is not acceptable to ACE, it can simply be rewritten by rearranging its words or stating ambiguous sections clearly. Another advantage of ACE is that it has tools that can convert the language into Discourse Representation Structure (DRS), which is a variant of first-order logic and a step closer to the form needed to be fed to an ATP [13]. DRS, and the theory behind it, are discussed later in this chapter.

2.2.3   CELT

Another CNL is Controlled English Logic Translation (CELT)[9]. It offers a powerful controlled English language and a set of useful options that differ from those offered by ACE. Although inspired by ACE, the main difference between ACE and CELT is that CELT is not only a language. It is more-so a tool that translates a restricted form of English into a first-order logic form [9]. In other words, CELT's main purpose is to perform syntactic and semantic analysis on the restricted English input, and then transform it into logic [9]. It does this by using terms that exist in SUMO. CELT is also mapped to WordNet's[8] extensive lexicon of over 100,000 words and word senses. Having WordNet and SUMO as a part of its translation engine makes CELT a unique system where each translated term has a relation to other words and meanings as described in SUMO.

Since CELT is aligned with WordNet, it is aware of an abundant amount of vocabulary. The controlled language used for CELT is limited by a strict set of grammar rules that is designed to avoid ambiguity, just like ACE. CELT uses mappings between words/word-phrases and WordNet/SUMO to translate into logic. CELT can translate present tenses and singular nouns, although it can use WordNet to fix some other tenses and plural verbs. CELT's translation capabilities are expected to change in future releases [9].

The logic form that CELT translates to is called Knowledge Interchange Format (KIF) [14]. The KIF language was developed by Michael Genesereth, a professor at Stanford University, in the early 1990s. The logic form is a variant of first-order logic.

---

[8]http://wordnet.princeton.edu/

It can represent knowledge and then share it across different types of programming languages [14]. CELT specifically translates to the Standard Upper Ontology KIF (SUO-KIF) [15], which is a variant of the KIF language, and also the language in which SUMO is written. SUO-KIF was specifically designed to interact with SUMO, and was designed for use in the authoring and interchange of knowledge [15]. Here are some examples of valid CELT sentences that can be translated into SUO-KIF:

Example 2.26 *Mary's car runs into the river bank.*

Example 2.27 *Henry the Eighth rules England.*

The power of SUMO and WordNet allows sentences like Example 2.26 and Example 2.27 to be understood without having multiple interpretations. In CELT, one can mention 'Henry the Eighth' as a person, where contrastingly ACE would not be able to do so and might interpret the phrase entirely wrong. The reason CELT can recognize 'Henry the Eighth' is because it exists as an entity in the SUMO ontology. Similarly, in Example 2.26, the concept of 'river bank' is a single notion in CELT (as it should be), while in ACE it is not. To express 'river bank' or 'Henry the Eighth' as a single concept in ACE, one must add hyphens:

Example 2.28 *Henry-the-Eighth rules England.*

Example 2.29 *Mary's car runs into the river-bank.*

Because of its similarity to ACE, CELT was one of the CNLs considered as an option for the query translation tool for CNL-WKR. Unfortunately, due to some functionality issues with the tool, it was later abandoned for ACE. The reason CELT was

abandoned was because of its limitations with translating complex sentences. CELT is capable of converting to TPTP, but fails to do so for numerous examples, even simple ones.

2.2.4 Discourse Representation Structure

One of the goals of CNL-WKR was to be able to translate from ACE to TPTP first-order logic. When translating a written language to first-order logic, meaning can often be lost. While the loss of meaning can be subtle, it still can have profound effects on the resulting translation. Since ACE offers a tool that translates ACE to Discourse Representation Structure (DRS) [16], which is a first-order logic variant that can retain the entire meaning of the original language, it was decided that the ACE translation should be done from DRS to TPTP rather than from ACE.

Trying to find a representation language for natural language that retains its original meaning has long been a topic of interest. For expressing single phrases, first-order logic is enough, but for expressing a sequence of related sentences, problems frequently arise [16]. A common translation problem is deciding how indefinite nouns should be represented. *A dog* is easy to understand as *one* dog when written in a sentence that is meant to refer to one dog:

Example 2.30 *A dog barks.*

Example 2.30 is not so clear when the words *not* or *every* are used within similar sentence structures:

Example 2.31 *A dog doesn't bark.*

or

Example 2.32 *Every man owns a dog.*

The confusion in Example 2.31 occurs when trying to think of it in terms of existential quantifiers. In logic, Example 2.31 can be expressed by saying "There does not exist a dog that barks." But that would consider all dogs, not just one. In Example 2.32, it is not clear whether the sentence is stating that all men each own their own separate dog or if all men have ownership of the same one dog. DRS can deal with issues like these as well as that of the unbound anaphora. An anaphora is an instance in a sentence that refers to another part of the sentence or another sentence entirely. For example, take the classic Donkey Sentence in the following example:

Example 2.33 *Every farmer that owns a donkey beats it.*

Example 2.33 is similar to Example 2.20 in the sense that it can carry an ambiguous interpretation. The word *it* is the anaphora here, since it is used to describe something that was mentioned earlier in the sentence, the donkey. What makes these anaphora difficult to interpret is the fact that it becomes vague what *it* is referring to: a single donkey, or all donkeys that are owned by a farmer?

Discourse Representation Theory (DRT), which is the theory behind DRS, was first developed by Hans Kamp in 1981 [16], in an effort to solve the anaphora problem shown in Example 2.33. Despite the theory's original intent, it is now widely used as a way of representing natural language involving dynamic interpretation [16]. Because of its ability to deal with complex grammars that yield ambiguous interpretations, DRS can be more expressive than other logic representations [16].

The idea behind DRT is that it takes a discourse (i.e., a number of sentences that come in sequence) and converts them to a DRS form. DRS has the ability to represent a complex discourse that introduces new entities (e.g., people, animals, objects, etc.) at different points in a discourse. DRS can also easily represent discourses that have references to entities that were introduced in earlier parts of sentences, much like the Donkey Sentence in Example 2.33.

Because the SPASS-XDB theorem prover does not use DRS as its input (it uses TPTP - see Section 2.1), DRS is used as an intermediate step before fully converting to TPTP. Since ACE already has the tools in place to convert to DRS, it is used a spring-board to convert to TPTP.

2.2.5   Translation of Natural Language to Logic

There are many logic formalizations of natural language, and not all of them are easy to translate into from the original natural language. In Section 2.1, the two levels of logic that were discussed were propositional logic and first-order logic.

Translating natural language statements to propositional logic is trivial, as it involves identifying connective words within the statements, such as *if*, *then*, *and*, *or*, and *not*, and replacing them with the appropriate mathematical symbol between statements. For example:

Example 2.34 *Either all men are created equal or grass is green.*

This can be translated into:

'All men are created equal' ∨ 'grass is green'

Despite propositional logic being trivial to translate, it is very limited in what it can say. Notice that in Example 2.34, using propositional logic for translation does not preserve exactly what is said. The fact that the left-hand side of the disjunction says something about the quantity of something (i.e., all men) suggests that one needs a more expressive logic to capture that notion.

For more expressivity, first-order logic or higher-order logics are required. To translate natural language statements into first-order logic, one must first determine which parts of the statements should be translated to predicates and which parts should be translated to functors. To do this one must decide which words indicate relationships, which words describe objects, and which describe actions or properties. As is discussed in Chapter 3, DRS provides all this information and makes translation from ACE to TPTP a lot easier to do than if the DRS were not given. The specific algorithms used to do this translation are discussed in the Chapter 3.

Natural language can also be translated to higher-order logics, but the task becomes increasingly difficult. Even though the languages become increasingly expressive, their usefulness to real-world applications (such as in the semantic web or theorem proving) decreases. Thus, the discussion and explanation of higher-order logic and their translations to and from natural language is beyond the scope of this research.

2.3 Natural Language Interfaces

To have a WKR system that is complete, it must have a user-friendly interface, specifically a good natural language interface (NLI). The whole purpose of being able to reason over world knowledge is so that queries can be submitted to a WKR system by the general public, rather than just people in computer-related fields. For the general public to be able to appreciate a WKR system, they must be allowed to use their own natural language (or a similar language) in conjunction with the system. The front-end of a WKR system needs to be able to take a user's natural language and convert that to a language that the WKR can understand. In other words, WKR systems need to provide casual end-users with the capability of querying a system without requiring them to learn the language that the underlying system uses [17].

NLIs offer users a simple way of formulating queries, while hiding the formalized languages used by ontologies and external sources [17]. Current WKR systems typically lack a means of making the human-computer interaction easy and simple. For a number of applications requiring human-computer interaction, it would be extremely desirable for computers to understand the natural languages that are spoken by their human users; NLIs address this issue. NLIs are best suited for their use in applications where the user's personal or financial cost of learning a formalized query language may outsize the user's desire to find the answer to a query.

In Section 2.2 the language translation process was discussed. In this section, the interfaces used to interpret the language will be focused on. First, this section will briefly discuss the history of NLIs. Then, it will discuss current language interfaces

and translators, with an emphasis on the interface tools offered by ACE.

2.3.1 Early Interfaces

Interest in NLIs first came about in the late 1960s [18]. One of the earliest interfaces was Lunar-9. It was an NLI that linked to a database that contained information pertaining to chemical analyses of moon rocks that were being retrieved during Apollo missions [18]. These first interfaces were very common at the time and typically had limited capabilities, and if they were linked to a database, the domain of that database was usually small and constrained.

One of the most well-known NLIs was SHRDLU, created by Terry Winograd in the late 1960s [18]. The system worked in a virtual environment that dealt only with blocks and pyramids. The system was able to perform actions on the items upon the user's command. The system could also answer questions about the motives of its actions and about the past and present layout of the items [18]. The user could make commands and queries like 'Pick up a big red block.', 'Are there any blue blocks in the box?', or 'Why did you pick up the green block?'. SHRDLU could also be given some information such as 'A steeple is a stack containing two green cubes and a pyramid.' and SHRDLU would from then on understand what a steeple was [18]. SHRDLU boasted having one of the first impressive interfaces for submitting natural language to a computer. While the allowable commands were limited, SHRDLU was very good at dealing with parts of language that are typically difficult to understand (for example, pronouns).

Other NLIs that existed early on in their history were ELIZA, and the chat bot ALICE [18]. Both were able to sustain full conversations with human users in pure English, but neither of them could converse deeply or provide deep or relevant insight. Chat bots like these only employed AI techniques that gave the *impression* of understanding language rather than truly understanding the language in a way that it could then be reasoned over. These two systems worked relatively well, but offered no useful application since they didn't do much with the user's input, other than attempt a response.

Since the 1960s, the progress and success of NLIs has been mediocre. Numerous NLIs were released throughout the 70s and 80s (e.g., Ladder, Chat-80, Janus), but none were extremely impressive or radically different from one another [19]. As a result, the interest in NLIs died down in the 1990s [19]. The necessity for robust and successful NLIs became more acute in the early 2000s as the amount of accessible world knowledge and the desire for querying about that knowledge increased. Also, the need for NLIs increased as the number of nontechnical users that wanted to access a wide range of databases through their web browsers, also increased [20]. Systems that linked to large databases (e.g., Wikipedia[9], Freebase[10]) containing information useful to users, began developing user-friendly NLIs to go along with those databases [19]. The rising popularity of the semantic web also added to the resurgent interest in NLIs. Rather then just having NLIs that attempted to interpret a user's input, NLIs were now being used to also retrieve world knowledge and deliver them back to

---

[9]`http://www.wikipedia.org`
[10]`http://www.freebase.com`

the user [17].

A few examples of such semantic web interfaces are ORAKEL and Squirrel [17]. ORAKEL is an NLI that is linked to an ontology and is capable of answering user queries. In relation to the lexicon used to access database information in ORAKEL, studies were shown that "people without any NLI expertise could adapt to ORAKEL by generating a domain-specific lexicon in an iterative process." [17]. Squirrel, on the other hand, is a search and browse interface that is linked to semantic data [17]. Users enter terms and can then search the results for their desired answer. This is similar to the idea of a search engine. Search engines have a rudimentary interface - an input field and no specific required query language. These interfaces are obviously easy for the user to learn and adapt to, but to extract results is more labor intensive.

In more recent years there has been a growth in powerful reasoners. For a reasoner to be useful, it is essential that it have a user-friendly NLI. There currently exist a great number of user-interfaces that are linked to databases, and even some that can perform reasoning. True Knowledge and Wolfram Alpha are two of those systems. True Knowledge can perform quite a bit of complex reasoning and can answer a large set of queries. Their NLIs are user-friendly and can accept natural language inputs and understand it to a limited extent. Wolfram Alpha is one of those systems and it can answer queries related to math problems as well as basic facts about people, objects, and even countries. Increasingly more WKR systems are being developed and their NLIs are becoming increasingly more powerful.

2.3.2 ACE Interfaces

ACE offers a large set of NLIs. Even though the ACE NLIs use the ACE language (which is a controlled language and not a pure natural language), a lot can still be accomplished and they are quite user-friendly. The ACE NLIs can be used easily for a variety of tasks, but they are more geared towards being used as tools or additions onto other systems [12].

The first ACE NLI, and probably the most relevant to this research is APE. APE is the ACE parsing engine. It is an easy to use interface where an ACE text is entered into an input field and the interface then converts the language into an alternative representation like DRS, OWL, or pretty print [12]. The interface is also capable of checking if a sentence is a properly formed ACE sentence and if not, where the errors lie. It also offers a very useful web service, which is used for this research to obtain ACE translations to DRS. As stated before, the interface does not do any query answering, but it does have the capability of understanding the ACE language and returning a different form of it back to the user.

Another useful NLI is the ACE Reasoner (RACE) [12]. The purpose of this interface tool is to reason over ACE texts. Users can enter a number of axioms and then ask queries about that set of axioms. Using RACE, the system can either prove or answer the given queries. RACE has three deduction modes: one that can check the consistency of a set of ACE axioms, one that can show that ACE theorems can be deduced from a set of ACE axioms, and one that can answer ACE queries from a set of ACE axioms [12]. Unfortunately, the system is not fully functional yet and

the examples provided on the ACE web site only show its limited capabilities. RACE can only perform reasoning on simple ACE texts. It can not reason about complex things where real, live pieces of world knowledge are needed.

ACE also offers the ACE Wiki Interface [12]. ACE Wiki allows users to enter ACE sentences into a database. These sentences either describe facts about the world or relations between things. It uses the OWL reasoner, Pellet[11], to ensure that the content remains consistent [12]. The interface is guided to ensure that correct ACE sentences are constructed.

ACE View is another interface that uses the ACE language. It can create ontologies and rulesets. The nice thing about this interface is that it allows users to create OWL knowledge-bases without having to know OWL syntax (the user can just use the ACE language instead). User's can also open up existing ontologies and view them in the ACE language [12].

2.3.3   Difficulties with NLI

The difficulties with creating an NLI that is effective and efficient are many. First of all, dealing with natural language has always been infamously difficult, so developing a powerful NLI takes a lot of time and is extremely difficult as a result. The best approach to this problem has been to use a CNL, or even a user-guided interface with pull-down menus [17]. Another obstacle for NLIs is that of their usefulness. Even if an NLI is very effective, it is never certain whether it will be accepted by the end-users or not. It is thought that "in the time of Google and graphical user-interfaces, where

---

[11]`http://pellet.owldl.com/`

people are used to formulating their information needs with keywords and then browse through dozens of answers to find the appropriate one or to clicking through menus and graphically displayed functions, full-fledged NLIs may be redundant." [17].

Another issue with NLIs is that users commonly ask complex queries or give complex commands that are far beyond the capability of the system to be able to interpret. On the other hand, if a the system uses a guided menu system, the user could only select options that are constrained to the limits of the system (e.g., AleXSI[12]).

2.4   World Knowledge Reasoning

World knowledge, as mentioned in Chapter 1, is a term used to represent knowledge about the world. Having ready access to world knowledge has always proven to be of importance. To have resources that can provide answers to queries about the world in a matter of minutes, if not seconds, allows people to function more efficiently in their every day lives, whether it be in a social or business setting. Current systems that provide world knowledge, like search engines or large knowledge-bases, can retrieve it relatively quickly. Being able to automatically reason over this knowledge though, has not yet been done on a large scale. Attempts have been made, but the systems have either been slow or covered only a small domain of knowledge.

This section first discusses the history of world knowledge retrieval and then describes a few systems that currently attempt to do reasoning over this knowledge.

---

[12]http://www.cs.miami.edu/~tptp/cgi-bin/AleXSI

2.4.1   Search Engines

One of the most widely used automated sources of knowledge retrieval are search engines. Ever since 1994, when one of the first web search engines, the World Wide Web Worm (WWWW), was created, the web has been growing rapidly [21]. Old web search engines like these had indices for, in pale comparison to current standards, a small number of web pages and documents available on the web. The number of queries given to these early search engines was small but grew from around a thousand a day in the late 90s, to about several hundred million a day over a span of about 10 years [22]. To keep up with the constantly growing web content, search engines began to employ better tactics to provide faster results. Storage space became larger and was used more efficiently. The indexing systems began to process data quicker and more efficiently as well. Since the creation of more recent search engines and the rise of Google, search engines have been able to cope well with web growth and most importantly, keep up with the ever growing world knowledge that is available online [22].

The drawback with search engines is that they do not provide answers. Instead, they provide sources which don't always contain answers. Knowledge retrieval isn't as simple as entering a query and getting back a single direct answer. Knowledge still must be retrieved. Improvements are constantly being made to search engines to increase their speed and to make their results "smarter," but the fact remains that search engines still do not provide direct answers to complex questions [23] . Without being able to provide direct answers, search engines are unable to reason because they

do not collect any world knowledge with which to reason over.

In the 90s, the philosophy behind search-engines was one that pushed the idea that the purpose of search-engines was to make web navigation easier [23]. In more recent years though, the philosophy has turned to one that expects search engines to provide direct answers, rather than act as a hub for easy navigation. People know they can find the information they want, but it has reached the point where people don't necessarily want to spend a lot of time searching through all the information to find what they want.

2.4.2 Knowledge-Based Systems

A substantial improvement towards WKR comes in the form of Knowledge-based Systems. Knowledge-based systems are systems that are built upon large databases of facts. These systems can perform database lookups and basic reasoning [1]. The main purpose of these systems is to answer queries directly. An obvious drawback is that they rely on hard-coded facts that have been "hand-made" or curated from the web [1]. This is a drawback because it takes a lot of time to create a substantial and useful knowledge-base. Another drawback is that a knowledge-base is typically made up of static knowledge. Knowledge needs to be constantly updated as things change in the world. To keep track of all the knowledge that has changed or that has the potential to change, is an arduous task and makes knowledge-bases less effective than initially thought. Also, answers to queries can only be found if facts about the query already exist in the knowledge-base (i.e., the knowledge-base already "knows"

about it).

There are a number of knowledge-base systems that currently exist and are capable of offering knowledge from which to reason over. A few examples that are discussed here are Cyc, SUMO, Wolfram Alpha, and True Knowledge.

Cyc

Cyc is a large knowledge-base that has a large amount of human knowledge stored in a formalized representation [1]. Knowledge, in this sense, includes things that are necessary for a computer to appear as though it has "common sense." This means having hard-coded factual knowledge, rules of thumb, and information on how to reason about everyday things [1]. Cyc's knowledge-base is made up of terms and assertions that either provide factual or ontological information and contains about 5 million assertions that describe relations between 500,000 individual terms [1]. Cyc uses a special formalized language, called CycL, to represent its world knowledge [1].

While a lot of the initial assertions and terms were handcrafted in the early 90s, new assertions are constantly being added by both an automatic procedure that scours the web and converts knowledge into assertions (with limitations), and manually [1]. Additionally, Cyc can add a great number of other assertions to itself by using an inference engine to infer new knowledge from pre-existing knowledge [1].

The knowledge-base that Cyc uses, is divided into different domains of knowledge, which each have a varying amount of detail and pertain to a particular interval of time. The purpose of this is to allow Cyc to compartmentalize its facts and help maintain a knowledge-base that is free from contradictions [1]. It also increases the performance

of the system since it can then focus on one specific part of the knowledge-base rather than the whole, if necessary.

SUMO

Another example of a large database of knowledge that can be reasoned over is the Suggested Upper Model Ontology (SUMO) [3]. SUMO is used as CNL-WKR's internal source of ontological axioms.

SUMO claims to be able to reason over axiomatized world knowledge and exhibit common sense. The axiomatized knowledge of SUMO ranges from general information to domain-specific concepts [3]. SUMO knows for example that humans are mammals and that mammals are animals or that humans communicate by talking. SUMO has around 20,000 terms and 70,000 axioms and is the largest open source formal upper ontology available [24]. SUMO has a useful online engineering environment, SigmaKEE[13], that lets users browse SUMO terms and hierarchies, WordNet mappings, and even proof results from SUMO axiom inferencing.

Like all databases though, the domains of knowledge that it contains are limited. SUMO has a tremendous amount of knowledge about cities and actors, for example, but lacks information about many other domains. Capturing the vastness of knowledge that exists in the world is nearly impossible, and while SUMO contains many complex facts about certain domains of knowledge, it lacks a large number of simple facts about the world. A solution to this problem has been approached by using an additional source of facts: YAGO, another large ontology [25]. The YAGO ontology

---

[13]http://sigmakee.sourceforge.net/

is one of the largest resources of factual knowledge available today [24]. It uses curated data from Wikipedia, the popular online encyclopedia, and combines it with the WordNet lexical database [25]. While, WordNet is powerful on its own, YAGO has added knowledge to the content of WordNet by taking information from Wikipedia [25]. YAGO contains facts about movies, actors, countries, and more and in total, contains more than one million single objects and more than 14 million facts about those objects [25]. One of the drawbacks of YAGO is that the ontological knowledge that it contains is limited and basic, so it only allows for very simple reasoning.

YAGO and SUMO have since come together to create an even greater source of axiomatized knowledge [24]. The combination of SUMO with YAGO has given a tremendous compendium of information that combines the formalization knowledge in SUMO with the huge body of knowledge gathered by YAGO. The way the two knowledge sources were combined was by recreating the content of YAGO into the structure of SUMO. Merging the two together created an ontology of roughly two million objects and several million axioms about the relations between them [24]. This obviously increased the number of entities in SUMO by a great number.

Wolfram Alpha

Wolfram Alpha is a recent project developed by Wolfram Research[14], the creators of the popular mathematics suite, Mathematica. It was released May 2009 and his been garnering a lot of interest ever since. It has also helped popularize interest in the future role that search engines will play in the world. Although Wolfram Alpha does

---

[14]`http://www.wolfram.com`. All information in this section was pulled from their web site.

not use much reasoning, it is capable of providing users with a lot of world knowledge facts. Wolfram currently stores hundred of domains of knowledge facts, including historical weather, drug data, currency conversion and other financial information, and more.

Unlike SUMO and Cyc, but similar to SPASS-XDB, Wolfram Alpha retrieves its world knowledge from external sources on the web which is then curated and stored. Another thing that is interesting about Wolfram Alpha is that it is specifically designed to be used by the general public. Similar in style to a search engine, the web interface consists of a single-line query box. The queries that can be submitted are restricted to the domains that are available from the external sources and most also be loosely written (i.e., lacking in grammar). One of the negative things about Wolfram Alpha is that it can not answer complex questions that require reasoning; it can only retrieve world knowledge facts and if desired, perform mathematical calculations on them.

True Knowledge

The True Knowledge[15] system was released a few years ago in beta form, and in early 2010 became live to the public. It claims to be the world's first answering engine. Much like Wolfram Alpha, the system is designed for public use. Its web site design is very user-friendly and also consists of a single-line query box where queries can be asked in any fashion. The system draws upon a large database of stored world knowledge. These facts have been curated and hand-picked from various sources on

---

[15]`http://www.trueknowledge.com`. All information in this section was pulled from their web site.

the web. The system claims to have over 100 million curated pieces of relational knowledge pertaining to millions of different things. A lot is kept secret about the architecture of the system, but what is known is that it does do some reasoning at some level. It can answer a decent number of queries and its running speed is fast, given a simple query. It still has difficulty answering complex queries and the domains of world knowledge that it covers is nowhere near complete.

Issues with Knowledge-bases

One of the issues with knowledge-bases and using them for WKR is that capturing a lot of world knowledge in one large database is extremely difficult and time consuming to create and maintain. For example Cyc, which claims to be a "common sense" reasoner, boasts having millions of "common sense" facts in its database but still can not answer a large number of queries. Cyc's database of facts will always need to be updated as things in the world change and progress. There is also the concern, especially with Cyc, as to how many world knowledge facts would be considered enough to cover *all* world knowledge? Can common sense really boil down to having a finite number of axioms?

A secondary problem with having large databases of facts to reason over is that dealing with large amounts of possible axioms is not efficient. To prove a lot of simple queries, one may only need a select few axioms. To find the right ones among thousands, if not millions, of axioms might take quite some time. One of the desired features of a WKR system is speed. Having to find the right axioms from an extremely large pool decreases the speed and efficiency of a system.

CNL-WKR

A solution that deals with the issues with knowledge-bases is to have a system which utilizes a theorem prover that can fetch specific pieces of world knowledge as they are needed from external sources. The CNL-WKR system is such a system. The theorem prover that the CNL-XDB system uses is SPASS-XDB, which is an amended version of another well-known first-order automated theorem prover, SPASS [26]. The CNL-WKR system is made up of SPASS-XDB, the internal SUMO ontology, and mediators which speak with external sources that provide world knowledge facts in the form of axioms back to SPASS-XDB [26]. The system relies on factual world knowledge that is already documented and available on the web and in online accessible databases, as well as ontological world knowledge available from SUMO. The CNL-WKR system is capable of fetching axioms from the external sources at run-time [26].

The CNL-WKR system, as opposed to the previous systems mentioned in this section, has numerous advantages. One advantage is that the language it reasons with is a highly expressive logic (first-order logic). Many other systems use some type of weaker logic, such as Description Logic [4]. Another advantage is that that the system does not need to store millions and millions of facts in its database, since it has a large ontology already built in and external sources to provide them. The details of how SPASS-XDB works and what the architecture of the CNL-WKR system looks like, is discussed in Chapter 3.

The CNL-WKR system has a large number of advantages over the previous systems mentioned in this section and can already answer some complex queries in a relatively

short amount of time. Therefore, the system's future looks promising.

2.5 Chapter 2 Summary

This concludes the literature survey. This chapter has discussed the process and history of natural language translation into logic, the history and current status of NLIs, and finally the history of WKR systems. The next chapter discusses the specifics of the research done for this thesis, specifically, creating an end-to-end WKR system.

Chapter 3

CNL-WKR Design and Implementation

This Chapter and Chapter 4 discusses the WKR system designed and used for this research. This chapter specifically discusses the system design as well as the method of implementation. Section 3.1 discusses the architecture of the CNL-WKR system and provides more detail about the overall process that is involved in carrying a user-query from the front-end of the CNL-WKR system all the way through to the back-end, resulting in an answer or proof. Section 3.2 discusses the SPASS-XDB theorem prover that lies at the heart of the system and how it performs reasoning. Section 3.3 describes the first important part of this research: the algorithm used to convert the DRS representation of an ACE query into TPTP syntax first-order logic which is then fed to SPASS-XDB. Next, Section 3.4.1 explains how TPTP axioms and conjectures are aligned with SUMO and/or the external sources in order to answer queries that rely on those sources for world knowledge. Section 3.4 goes into detail on how mediators are used to speak between SPASS-XDB and the external sources that provide world knowledge. Section 3.5 discusses the implementation of the Perl script that runs the entire CNL-WKR system from end to end and how the web interface

uses it. Finally, Section 3.6 provides a brief summary of the chapter.

3.1 Architecture

Starting from the front-end, a WKR system needs some type of user-interface where a user can enter their query or a set of statements that can then be submitted to the theorem prover. ACE was the query language of choice.

To start using the CNL-WKR system, the user must submit an acceptable query in the ACE language. This input is then translated into TPTP first-order logic. The algorithms used to do this are discussed in Section 3.3. The ACE query is first converted to DRS via the available tools provided by the creators of ACE. Next, the DRS is passed on to NACE, which is the tool that converts DRS to TPTP. While TPTP is the necessary format required for reasoning with SPASS-XDB, the TPTP (depending on its intended use) must be aware of the syntax of external sources, ontologies, and mediators that are available to fetch the appropriate world knowledge. To do this, the TPTP can be sent through the SUMO/mediator alignment tool `get_Includes`, which adds the necessary axioms to the initial query depending on what the subject of the query is. For example, if a query about weather is submitted, the alignment tool will add axioms related to weather so that the Weather mediator can be used.

Once the additional axioms are added, the axioms and the conjecture are passed through the pretty-printing tool, TPTP4X, which prepares and organizes the axioms and conjecture into the appropriate form required by SPASS-XDB. Once this is complete, the translation from a user-input to TPTP is considered complete and the

query is now ready to be passed onto the core of the CNL-WKR system, SPASS-XDB. Once SPASS-XDB has fetched the axioms needed to attempt to answer a given query, and it has begun running its inference process, a proof (or answer) is hopefully found. Within a found proof lies the answer to a query. In certain cases it might be desired to have the full proof provided as the output and in other cases the answer alone is enough; the system provides options for both cases.

The overall process of the CNL-WKR system is illustrated in Figure 3.1. There are three main partitions that are highlighted, as they are the result of this research. The make up of these three partitions are described in detail in the following subsections. The partitions are:

1. Pre-processing: converting from ACE to TPTP

2. Alignment: aligning the TPTP with the external/internal sources

3. Reasoner and Proof: reasoning using the internal/external sources

3.2 The SPASS-XDB Theorem Prover

SPASS-XDB is the reasoner used at the core of the CNL-WKR system. It is a modified version of the well-known theorem prover, SPASS [26]. When the reasoner is called in the the CNL-WKR system, it receives a problem file, written in TPTP format, that contains information about the specifications for external sources of world knowledge, internal axioms provided by the SUMO ontology, and the query (or conjecture) to be answered. The specifications that are provided to SPASS-XDB about the external
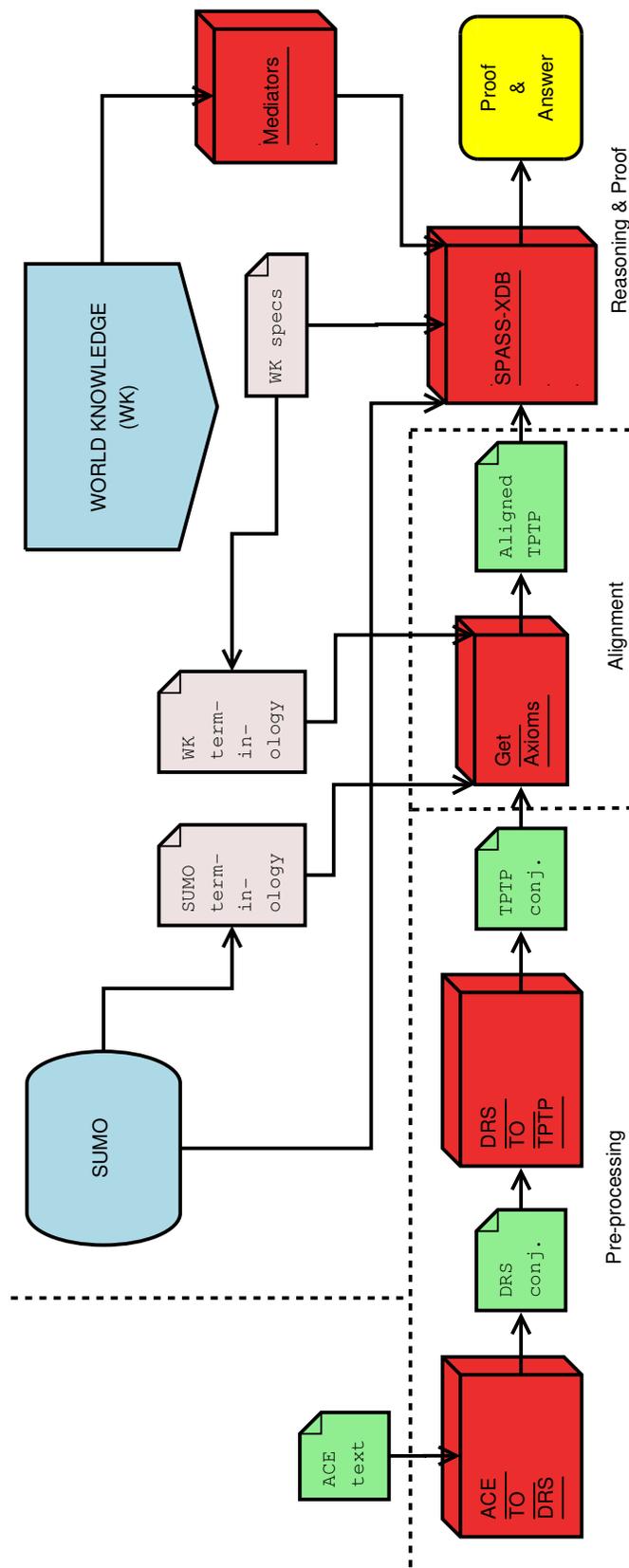
Figure 3.1: Overall process of the CNL-WKR system

sources details information about the names of the available executable mediators. It also provides information about controlling the information provided by the external source [26].

The algorithm that SPASS-XDB uses is based on the algorithm used by SPASS. It is the classic Given-clause algorithm which is described in the following algorithm [27]:

Algorithm 1 Given-clause Algorithm

1: Let CanBeUsed = ∅
2: Let ToBeUsed = Input clauses
3: **while** Refutation has not been found & ToBeUsed not empty **do**
4:     Select first item in ToBeUsed (i.e., the ChosenClause)
5:     Move the ChosenClause to CanBeUsed
6:     Infer all possible clauses using the ChosenClause and other clauses from Can-
       BeUsed
7:     Add the inferred clauses to ToBeUsed
8: **end while**

SPASS-XDB differs from the original SPASS by the fact that it can call on the external sources for axioms that are needed at run time during the deduction process. In Algorithm 1, the ToBeUsed clauses are the conjecture (i.e., query) and any other additional axioms that were added by the alignment tool. First, the ChosenClause (which is the first item in the ToBeUsed clause list), is selected and moved to the CanBeUsed list. Here SPASS-XDB attempts to infer all possible clauses using the ChosenClause and any other clauses in the CanBeUsed list. While this is happening, axioms provided by the external sources (or world knowledge facts) are requested when a negative literal of the ChosenClause matches the form of an external specification [26]. The retrieved axioms are then added to the ToBeUsed list. While the external sources are being called, inference simultaneously continues between the

ChosenClause and other clauses from the CanBeUsed list. Here is where SUMO axioms lie and are used (since they are an internal source), if there are any to be used. Finally, if a refutation has been found, the proof is returned.

SPASS-XDB is just as sound as SPASS. Its completeness though, is somewhat of a delicate issue. SPASS was originally designed to attempt to solve problems where all the axioms and the conjecture would be included in the initial problem file. This is not the case for SPASS-XDB as there are axioms that can be added to the set of clauses during run time. In order for SPASS-XDB to remain a system that is as complete as possible, certain constraints of the original SPASS system had to be relaxed [26]. The problem with relaxing constraints, for one, is that it increases the search space. The search space can be affected by the number of available external axioms that can be fetched. Hence, SPASS-XDB uses certain techniques to control and focus the manor in which these external axioms may be retrieved and delivered.

3.3 Process of Converting DRS to TPTP

One of the tools provided by ACE (i.e., APE) is capable of converting the ACE language to DRS [28]. Since the ACE to DRS conversion is already provided, this research focusses on the conversion from DRS to TPTP first-order logic form using NACE. NACE is one of the major points of focus in this research as it is one of the key components to the user-friendly side of using the CNL-WKR system. This tool was written in the PROLOG language using SWI-PROLOG[1]. During the DRS to

---

[1] http://www.swi-prolog.org/

TPTP translation process, the elements contained in the DRS are first parsed, one by one. Each element is then processed and then depending on the element's part of speech, the appropriate predicate is created along with its existentially quantified or universally quantified variables. These predicates are then grouped together to create the final TPTP formula. More on the details of the structure of the DRS are explained in Section 3.3.1.

3.3.1 ACE to DRS

The details of the conversion from ACE to DRS that is provided by the ACE tools is beyond the scope of this research. What is discussed here is the basic form of the DRS that is a result of an ACE query. The DRS described in this section is based on queries written in version 6.5 of ACE [28]. The DRS that is returned is written in a notation where there is a single DRS predicate containing two arguments, the list of variables quantified and the list of predefined predicates that represent specific parts of speech and contain pieces of ACE text as their argument:

```
drs(Domain, Conditions)
```

The `Domain` argument is a list that contains any quantified variables in the ACE text. The `Conditions` argument is a list that contains all of the elements of the ACE text. For example:

Example 3.1

```
drs([], [=>(drs([A], [object(A, man, countable, na, eq, 1)-1/2]),
        drs([B, C], [object(B, human, countable, na, eq, 1)-1/5,
        predicate(C, be, A, B)-1/3]))])
```

In Example 3.1, ',', which separates elements in the `Conditions` list, represents the logical symbol of conjunction between each element [28]. If a DRS has to represent other logical connectives, other symbols are included in the list surrounding the elements. These include negation '`-`', disjunction '`v`', and implication '`=>`'. It should also be noted that each element in the `Conditions` list has a sentence and token number attached to the end of it. This first number refers to the sentence number that the item is in. The second number refers to the token position of the item in the specified sentence.

The possible elements in the `Conditions` list are the following: `object`, `property`, `relation`, `predicate`, `modifier_adv`, `modifier_pp`, `has_part`. The following definitions for each are provided:

`Object` predicates stand for nouns. They are formed with the following syntax:

```
object(Ref, Noun, Quantity, Unit, Operator, Amount)
```

`Ref` stands for the variable which represents the object at hand within the ACE text. `Noun` stands for the object itself. `Quantity` stands for the type of object it is in terms of quantity, i.e. an object is either `countable` or something that can only be measured by `mass` (for example, water is not countable, but its mass can be quantified). `Unit` represents the unit needed to measure the mass of an object that has the `Quantity` property equal to `mass`. `Operator` stands for measurements of an object, i.e., equal (`eq`), exactly (`exactly`), greater than (`greater`), greater than or equal to (`geq`), less than (`less`), less than or equal to (`leq`), or not applicable (`na`). Finally, `Amount` represents the numerical amount that exists of the object [28].

`Property` predicates stand for words in an ACE text that describe objects, otherwise known as adjectives. They are of the form:

```
property(Ref1, Adjective, Degree)
```

`Ref1` stands for the variable which represents the primary object which the adjective describes. `Adjective` stands for the adjective itself. `Degree` denotes the degree of the adjective. Since adjectives can either be positive, comparative, or superlative (e.g., 'thin,' 'thinner,' or 'thinnest'), `Degree` is used to specify this. `Degree` can either be `pos`, `pos_as`, `comp`, `comp_than`, `sup` [28].

`Relation` predicates are used to represent words that are linked by `of`. They are of the form:

```
relation(Ref1, of, Ref2)
```

`Ref1` stands for the variable which represents the object on the left side of the relation. `Ref2` stands for the variable which represents the object on the right side of the relation [28].

`Predicate` predicates are used to represent intransitive, transitive, and ditransitive verbs. They are of the form:

```
predicate(Ref, Verb, SubjectRef)
predicate(Ref, Verb, SubjectRef, ObjectRef)
predicate(Ref, Verb, SubjectRef, ObjectRef, IndirectObjectRef)
```

`Ref` stands for the variable that represents the verb. `Verb` is the verb itself. `SubjectRef` is the variable which represents the subject of the verb. `ObjectRef`

is the variable which represents the object of the verb. `IndirectObjectRef` is the variable which represents the indirect object of the verb [28].

`Modifier_adv` predicates are used to represent verb modifiers, also knows as adverbs. They are of the form:

    modifier_adv(Ref, Adverb, Degree)

`Ref` stands for the variable that represents the adverb as well as the verb being modified. `Adverb` represents the adverb itself. Since adverbs can either be positive, comparative, or superlative (e.g., 'quickly,' 'more quickly,' or 'most quickly'), `Degree` is used to specify this. `Degree` can either be `pos`, `comp`, and `sup` [28].

`Modifier_pp` predicates are used to represent prepositional phrases. They are of the form:

    modifier_pp(Ref1, Preposition, Ref2)

Much like the `relation` predicate, `modifier_pp` predicates links one thing to another. `Ref1` stands for the variable that represents the verb before the preposition. `Preposition` stands for the preposition itself. `Ref2` stands for the variable that represents the object that the preposition is linked to [28].

`Has_part` predicates are used to describe objects that have a membership to another group of objects. They are of the form:

    has_part(GroupRef, MemberRef)

`GroupRef` stands for the variable that refers to a group of objects. `MemberRef` stands for the variable that represents the object that belongs to the group [28].

When representing implications and disjunctions, DRS uses nested DRS predicates:

```
drs([],[drs([A],[condition(A)]) => drs([B],[condition(B)])])
drs([],[drs([A],[condition(A)]) v drs([B],[condition(B)])])
```

DRS has other capabilities, such as the ability to represent modal logic operators, questions, and command statements. They are not discussed in this research as they are not immediately relevant.

3.3.2   DRS to TPTP

Once the ACE text has been converted to DRS, NACE then takes the DRS and converts it to TPTP first-order logic. As discussed in Section 3.3.1, DRS has a finite number of possible list elements. Each of these has to be dealt with in a unique way. The basic idea of first-order logic translation is to locate things that are objects, things that describe relationships, and things that describe actions or properties. DRS provides a nice, ordered structure of all this information. What NACE does is picks out the elements of this information one by one and converts them to the correct TPTP predicate or functor. NACE also is aware of which variables must be existentially quantified and which must be universally quantified. Careful analysis of the scope of these variables is also done.

NACE

NACE was written in the PROLOG language using SWI-PROLOG. The main function of NACE is split up into a few sub-functions. The first few involve pre-processing the DRS. Although the DRS is already highly organized and contains a large amount of information about the submitted ACE text and its grammar, some alterations to the content and some extra ordering must occur before anything else is done.

First, there is a check done on the `Conditions` list of the DRS for disjunction and implication. Next a check is done on the contents of the `Conditions` list. Certain elements of the DRS list need to be adjusted in order to make the translation easier. This is described in more detail in the following subsections. Next, the list is ordered, placing `object` predicates at the highest priority. Next, this adjusted list is passed onto a function that removes each element from the list, converts it to a predicate or functor, and then adds it to a separate list. This newly created list of predicates and functors is finally passed to a function that conjoins all the elements of that list as well as adds any disjunction or implication symbols. It then adds the appropriate universal or existential quantifiers.

Checks

The pre-processing algorithm is described in Algorithm 2 and Algorithm 3. One is simply a continuation of the other and is to be viewed as one large algorithm. The algorithm takes an input of the predicate form `drs(D,C)`. C is a non-empty list that contains a finite number of elements, E.

First, a check on whether there are any nested DRS' must be done. This happens when a conditional or disjunction is used, like in Example 3.1. If either is found, the nested DRS is rewritten as a separate predicate (of the form `implies(A,B)` or `disjunct(A,B)`) that strips the nested DRS' of their respective, C lists. These C lists then become the argument of the newly created predicate. Any new elements created during this algorithm are then added back to the original C list and the old ones discarded.

After the nested DRS check, checks on certain DRS predicates are made. First, checks are done on elements in the form of `has_part/2` like in the following example:

Example 3.2

```
drs([A,B,C,D,E],[object(A,prize,countable,na,eq,1)-1/5,
        has_part(D,A)-1/'',predicate(B,win,E,D)-1/3,
        object(C,beer,countable,na,eq,1)-1/8,
        has_part(D,C)-1/'',
        object(D,na,countable,na,eq,2)-1/'',
        object(E,man,countable,na,eq,1)-1/2])
```

For translation, it makes more sense to get rid of these predicates altogether and simply replace any variables of elements that match the first argument of each `has_part/2` with the second. This is done so that all items that are of the same group have the same reference variable. So in this section of the algorithm, the C list is searched for any `has_part` elements. For each one found, another search is done on C, but this time to find all elements with reference variables that match the first variable of the `has_part` predicate at hand. Once a match is found, its reference variable is replaced with the second variable of the `has_part` predicate at hand.

Next, a check is done to remove predicates in the form of `predicate/5` like in the following example:

Example 3.3

```
drs([A,B,C,D],[object(A,money,countable,na,eq,1)-1/5,
          object(B,teller,countable,na,eq,1)-1/8,
          predicate(C,give,D,A,B)-1/3,
          object(D, man,countable,na,eq,1)-1/2])
```

These elements represent verbs that relate to an object, subject, and indirect object (e.g., *'The man gives money to the teller'*). Sentences that contain these verbs always use the preposition '`to`' to link the indirect object to the rest of the sentence. For some reason, ACE does not convert these prepositions to the form of other prepositions, `modifier_pp/3`. The purpose of this check is to turn these types of predicates into ones that only have an object and subject reference and then create a new preposition predicate that links the indirect object. So, for all `predicate/3` predicate elements in C, the last variable is stripped and a new `modifier_pp` is created with the preposition '`to`', and reference variables that reference the appropriate `predicate` predicate and `object` predicate included as arguments. So Example 3.3 would become the following:

Example 3.4

```
drs([A,B,C,D],[object(A,money,countable,na,eq,1)-1/5,
          object(B,teller,countable,na,eq,1)-1/8,
          predicate(C,give,D,A)-1/3,
          modifier_pp(C,give_to,D,A,B)-1/3,
          object(D,man,countable,na,eq,1)-1/2])
```

The next two checks are very similar. They involve searching the C list for predicates in the form of `modifier_pp/3` or `modifier_adv/3` like in the following examples:

Example 3.5

```
drs([A,B,C],[object(C,prize,countable,na,eq,1)-1/2,
             object(B,man,countable,na,eq,1)-1/6,
             predicate(A,be,C)-1/3,modifier_pp(A,for,B)-1/4])
```

Example 3.6

```
drs([A,B],[object(B,man,countable,na,eq,1)-1/2,
           predicate(A,win,B)-1/3,
           modifier_adv(A,quickly,pos)-1/4])
```

If found, the verb linked to it is then searched for in C. The verb word is then appended to the adverb or preposition. The reason for doing this is that if a preposition or adverb is to be represented as a logic predicate, it is not enough to simply have a predicate that names the adverb or preposition; the verb it modifies must also be indicated. So, the old `modifier_pp` or `modifier_adv` are discarded and new ones are created that indicate the modified preposition or adverb name as well indicate the reference variables provided by the verbs that are linked to them.

Next, all predicates in the form of `predicate/4` like the following example, where the verb at hand is 'be', must be removed.

Example 3.7

```
drs([A,B,C],[object(A,winner,countable,na,eq,1)-1/5,
             predicate(B,be,C,A)-1/3,
             object(C,man,countable,na,eq,1)-1/2])
```

C is searched for all elements that contain the reference variable subject reference variable. Each one that is found has that variable replaced with the object reference variable from the `predicate` predicate. The reason this is done is mainly for aesthetics. If left alone, the logic translation would create a predicate of the verb 'be'. Instead, though, the subject reference variable of the predicate is searched for in the C list and when an element is found, its reference variable is replaced with the object reference variable from the `predicate` predicate. This allows for no awkward predicates relating to the copula. Example 3.7 would become the following:

Example 3.8

```
drs([A,B,C],[object(C,winner,countable,na,eq,1)-1/5,
             object(C,man,countable,na,eq,1)-1/2])
```

Next, all predicates in the form of `relation/3` like the following example, must be removed.

Example 3.9

```
drs([A,B],[object(A,member,countable,na,eq,1)-1/4,
           object(B,family,countable,na,eq,1)-1/7,
           relation(A,of,B)-1/5])
```

They are replaced with a new `relation` predicate of the same form, but with the central 'of' replaced with the sentence token word that comes prior to (appended with 'of'). So, in Example 3.9, `of` would become `member_of`. The purpose of this is so that when the logic translation occurs, information about what word is being modified by the word 'of' is not lost.

Finally, before the pre-processing is finished, the C list is once again scanned for any predicates that have reference variables of the form `named(N)`, like in the following example:

Example 3.10   `drs([A],[predicate(A,win,named('Geoff'))-1/2])`

These predicates signify names of things. The predicate is stripped so that the name within the predicate is not enclosed within a predicate. In Example 3.10, `named('Geoff')` would become 'Geoff.'

Ordering

A *quicksort* algorithm is used to order the list of DRS elements. Special care is taken to order the elements specifically by order of importance. Importance is based on a simple principle: that objects need to be converted to logic first. The reason for this is that objects may be plural. If an object is plural (as is shown in Algorithm 4), duplicates of other elements in the list that refer to that object need to be created to represent each instance of the object. When the translation to logic occurs, a lot of the detailed information from the DRS is removed, such as the information related to plurality. Therefore, plurals must be dealt with before the translation occurs. So, `object` predicates are given the highest importance score of 3. `Predicate` predicates are given a score of 2. The remaining elements are given a score of 1.

DRS to Predicate List

The conversion from DRS list to predicate logic list is described in Algorithm 4. Once the C list is pre-processed and ordered, it is then passed through Algorithm 4. The

Algorithm 2 DRS Pre-processing Algorithm

Require: Input $= drs(D, C)$
Require: $C \neq \emptyset$
Require: $E \in C$
 1: **if** $v(drs(D_1, C_1), drs(D_2, C_2)) \in C$ **then**
 2:     Recurse to process C_1
 3:     Recurse to process C_2
 4:     Recursively rewrite $v(drs(D_1, C_1), drs(D_2, C_2))$ as $disjunct(C_1, C_2)$
 5: **end if**
 6: **if** $\Rightarrow (drs(D_1, C_1), drs(D_2, C_2)) \in C$ **then**
 7:     Recurse to process C_1
 8:     Recurse to process C_2
 9:     Recursively rewrite $v(drs(D_1, C_1), drs(D_2, C_2))$ as $implies(C_1, C_2)$
10: **end if**
11: **for all** $has\_part(Var_1, Var_2) \in C$ **do**
12:     **for all** $E \in (C \neq has\_part(Var_1, Var_2))$ **do**
13:         **if** $Var_1 \in E$ **then**
14:             replace Var_1 with Var_2 in E
15:         **end if**
16:     **end for**
17: **end for**
18: **for all** $predicate(Var_1, Verb, Var_2, Var_3, Var_4) \in C$ **do**
19:     create $predicate(Var_1, Verb, Var_2, Var_3)$
20:     create $modifier\_pp(Var_1, to, Var_4)$
21:     remove $predicate(Var_1, Verb, Var_2, Var_3, Var_4)$
22:     add $predicate(Var_1, Verb, Var_2, Var_3)$ to C
23:     add $modifier\_pp(Var_1, to, Var_4)$
24: **end for**
25: **for all** $modifier\_pp(Var_1, Prep, Var_2) \in C$ **do**
26:     **for all** $E \in (C \neq modifier\_pp(Var_1, Prep, Var_2))$ **do**
27:         **if** $E = predicate(Var_1, Verb, Var_3, Var_4)$ **then**
28:             $VerbPrep = Verb$ appended to $Prep$
29:             create $modifier\_pp(Var_1, VerbPrep, Var_3, Var_4, Var_2)$
30:             remove $modifier\_pp(Var_1, Prep, Var_2)$
31:             add $modifier\_pp(Var_1, VerbPrep, Var_3, Var_4, Var_2)$ to C
32:         **end if**
33:     **end for**
34: **end for**

Algorithm 3 DRS Pre-processing Algorithm Continued...

Require: Input $= drs(D, C)$
Require: $C \neq \emptyset$
Require: $E \in C$

 1: **for all** $modifier\_adv(Var_1, Adverb, Degree) \in C$ **do**
 2:    **for all** $E \in (C \neq modifier\_adv(Var_1, Adverb, Deg))$ **do**
 3:      **if** $E = predicate(Var_1, Verb, Var_3, Var_4)$ **then**
 4:         $VerbDegAdverb = Verb$ appended to Deg and $Adverb$
 5:         create $modifier\_adv(Var_1, VerbDegAdverb, Var_3, Var_4, Var_2)$
 6:         remove $modifier\_adv(Var_1, Adverb, Deg)$
 7:         add $modifier\_adv(Var_1, VerbDegAdverb, Var_3, Var_4, Var_2)$ to C
 8:      **end if**
 9:    **end for**
10: **end for**
11: **for all** $predicate(Var_1, be, Var_2, Var_3) \in C$ **do**
12:    **for all** $E \in (C \neq predicate(Var_1, be, Var_2, Var_3))$ **do**
13:      **if** $Var_2 \in E$ **then**
14:         replace Var_2 with Var_3 in E
15:      **end if**
16:    **end for**
17: **end for**
18: **for all** $relation(Var_1, of, Var_2) \in C$ **do**
19:    **for all** $E \in (C \neq relation(Var_1, of, Var_2))$ **do**
20:      locate $token$ in sentence previous to $relation$
21:      $NewRelation = token$ appended to of
22:      create $relation(Var_1, NewRelation, Var_2)$
23:      remove $relation(Var_1, of, Var_2)$
24:      add $relation(Var_1, NewRelation, Var_2)$ to C
25:    **end for**
26: **end for**
27: **for all** $E \in C$ **do**
28:    **if** $named(N) \in E$ **then**
29:      strip $named$ predicate
30:      replace with just N
31:    **end if**
32: **end for**

---

algorithm takes each element E in C and converts it to a predicate form. Each of these are then added to *LogicList*. Before the algorithm terminates, each `object` predicate is checked for plurality. If it is indeed a plural object, copies of the `object` predicate are made (with new reference variables, since they are new objects) and added to the C list. Also, the C list is searched for any other predicate that contain reference variables that refer to that same object. These predicates are also duplicated (with new reference variables as well) and added to the C list. The following show each type of DRS element and their corresponding converted logic predicate:

```
object(Ref, Noun, Quantity, Unit, Operator, Amount) --> noun(Ref)
predicate(Ref, Verb, SubjectRef) --> verb(Ref,SubjectRef)
predicate(Ref, Verb, SubjRef, ObjRef) --> verb(Ref,SubjRef,ObjRef)
property(Ref1, Adjective, Degree) --> degree_adjective(Ref1)
relation(Ref1, of, Ref2) --> token_of(Ref2)
modifier_pp(Ref1, Prep, Ref2) --> prep(Ref2)
modifier_pp(Ref1, Prep, Ref2, Ref3) --> prep(Ref2, Ref3)
modifier_adv(Ref1, Adverb, Ref2) --> adverb(Ref2)
modifier_adv(Ref1, Adverb, Ref2, Ref3) --> adverb(Ref2, Ref3)
```

Predicate List to Logic

Finally, to complete the translation from DRS to logic, the logic predicate list must be extracted and rewritten in the correct TPTP form, adding any existential or universal quantifiers wherever they are needed. This algorithm is described in Algorithm 5. As the algorithm cycles through the list of predicates, if the list does not contain `implies` or `disjunct` predicates, it simply takes each element and conjoins them. Then, all the variables contained in the conjoined formula are amassed and an existential operator is appended to the front of the formula, followed by the list of variables.

Algorithm 4 Ordered DRS to List Algorithm

Require: $C =$ Ordered DRS List
Require: $E \in C$
Require: $LogicList = \emptyset$

 1: **for all** $E \in C$ **do**
 2:     convert E to predicate form P
 3:     add P to $LogicList$
 4:     **if** $(E = object(Ref, Noun, Quantity, Unit, Operator, Amount)$ **then**
 5:       **if** $Amount > 1$ **then**
 6:         make $Amount$ copies of E with different Ref variables
 7:         make $Amount$ copies of all other elements in C that share Ref
 8:         add all copies to C
 9:         add predicate `look_different/2` for each combination of similar objects
           to C
10:       **end if**
11:     **end if**
12: **end for**

If an `implies` or `disjunct` predicate is encountered, the left-side elements, C_1, are conjoined together to create $Formula_1$. Next, the right-side elements, C_2 are conjoined together as well to create $Formula_2$. If the algorithm is dealing with `implies` it universalizes the list of variables from C_1 and appends itself to C_1. It then existentializes the list of variables from C_2 and appends itself to C_2. C_1, and C_2 are then connected by the implication operator (`=>`). If a `disjunct` predicate is encountered, C_1 and C_2 are connected by the disjunction operator (`|`). Then all variables are existentialized over. The final result is $Formula_f$.

Once a formula is returned by the DRS to TPTP converter, it is returned in the TPTP form:

```
fof(name,type,formula)
```

Where `name` is the name of the formula, `type` is the type of formula (`axiom` or `conjecture`), and `formula` is the actual logical formula. The name of the formula

---

Algorithm 5 Predicate List to Logic Algorithm

Require: $LogicList$ = list of predicates
Require: $E \in LogicList$

 1: **if** $(implies(C_1, C_2) \vee disjunct(C_1, C_2)) \in LogicList$ **then**
 2:   recurse to process C_1
 3:   recurse to process C_2
 4:   **for** each C_1 and C_2 recursively do the following **do**
 5:     conjoin all elements of C_1 to create $Formula_1$
 6:     conjoin all elements of C_2 to create $Formula_2$
 7:     extract all variables from $Formula_1$ into list V_1
 8:     extract all variables from $Formula_2$ into list V_1
 9:     **if** $(implies(C_1, C_2)$ **then**
10:       append $\forall$ to V_1 to get Var_1
11:       append Var_1 to $Formula_1$
12:       append $\exists$ to V_2 to get Var_2
13:       append Var_2 to $Formula_2$
14:       connect $Formula_1$ to $Formula_2$ via implication to get $Formula_f$
15:       **return** $Formula_f$
16:     **end if**
17:     **if** $disjunct(C_1, C_2)t$ **then**
18:       append $\exists$ to V_1 and V_2 to get Var
19:       connect $Formula_1$ to $Formula_2$ via disjunction to get $Formula_3$
20:       append Var to $Formula_3$ to get $Formula_f$
21:       **return** $Formula_f$
22:     **end if**
23:   **end for**
24: **else**
25:   conjoin all E together to create $Formula$
26:   extract all variables from $Formula$ into list V
27:   append $\exists$ to V and $Formula$
28:   **return** $Formula$
29: **end if**

---

is simple `name1`. Any axioms added later by `get_Includes` are numbered accordingly. All queries are given the `conjecture` type, and any axioms added later by `get_Includes` are given the `axiom` type.

3.4   Mediators and External Sources

SPASS-XDB has external sources from where it gathers real time world knowledge facts from the web and other places. There are two important parts to providing external world knowledge to SPASS-XDB. The first is deciding *which* external sources to include as sources of world knowledge facts. There are obviously many domains of world knowledge and so many external sources that provide them. One of the larger goals of a WKR system is to have access to *all* world knowledge (or as much as possible). To do this, a system must have a huge variety of external sources. If a system had an external source for each domain, then it would have complete access to all world knowledge. Since, that is hard to do (and is beyond the scope of this research), only a select few have been chosen for this research to test SPASS-XDB. The second important part is having a mediator that can interface between the theorem prover and the chosen external sources. The information provided by external sources can vary in form and notation, so different mediators need to be tailor-made for each external source.

The general structure of the mediators used in this research is relatively simple. Currently there are four types of mediators that have been implemented. The first is the Prolog mediator. This mediator is used to do basic procedures in Prolog,

such as arithmetic [26]. The next two mediators are the SQL and SPARQL mediators [26]. These are used to call on large knowledge bases such as DBPedia[2] and YAGOSUMO[24]. The last type of mediators, and the most relevant to this research, are the WWW mediators. These mediators are used to call on public web services. The code for all of these mediators is generic, and can be used with different external sources through small modifications. The following are the WWW mediators created for this research:

XChange

XChange was initially chosen because financial knowledge is a very popular domain. To have a mediator that can provide financial information is crucial. XChange currently provides axioms containing currency conversion data. The currency exchange rate data is obtained from Time Genie[3]. The mediator works by being sent a question that specifies a source currency and amount, and a target currency. The axioms created by the mediator then include the amount in the target currency. Time Genie updates the currency rates constantly.

AmazonBooks

Product knowledge is a popular domain. People constantly desire information pertaining to consumable products. Amazon[4] provides a large database of information about the products it sells. For this research, only information related to books is

---

[2] http://www.dbpedia.org
[3] http://rss.timegenie.com/
[4] http://www.amazon.com

used. Amazon provides a web service where requests can be sent and a large amount of data is returned in XML form. The mediator sends a question to Amazon in the form of a predicate that takes an author name, and returns information about all the books by that author (sold by Amazon) as well as information about each book. The information provided for each book includes: title, type of binding, publication date, and price in US dollars. More information is available, but was not used for this research.

Weather

Weather is a very important domain of world knowledge. Currently, there are two mediators available that speak to two different weather sources: the Weather mediator and the With_weather mediator. The Weather mediator uses GeoNames[5] as its source and the With_weather mediator uses Yahoo Weather[6]. GeoNames takes a given latitude and longitude and obtains the weather for those coordinates. It should be noted that GeoNames can only retrieve weather information for locations that have a publicly accessible weather station. Yahoo Weather is used in a slightly different manner. A weather condition (e.g., 'sunny,' 'partly cloudy') and country name are sent to the service and the cities that currently have that weather condition, are returned in an axiom form. This is done by cycling through the different location codes that Yahoo Weather offers for each country and finding a match. The speed of this mediator is dependent on the size of the given country, for there are more location codes to sift through if the country is large.

---

[5]`http://www.geonames.org`
[6]`http://weather.yahoo.com`

Location

The Location mediator uses data from GeoNames to provide geographical information about cities and places. GeoNames is a great source of geographic data as it can supply data ranging from weather, to population size, to elevation. The mediator sends out a question to the GeoNames web site in the form of a predicate that takes the name of a city or area. The latitude, longitude, and elevation are then delivered back as an axiom.

Other Mediators and the Future

Other external sources include the YAGO knowledge base [24], DBPedia[7], Mondial[8], and BabelFish[9]. Other mediators also exist that use the capabilities of Prolog. For example, there is a mediator that evaluates ground arithmetic expressions, another that provides information to whether or not two different atoms are syntactically unequal, another that provides regular expression matching capabilities, and also another that provides pretty printed formats of axioms.

Developing the AmazonBooks mediator so that it can retrieve information about all Amazon products is an obvious first step when considering expanding external sources in the future. Other future possibilities would be to use external sources that provide world knowledge about sports, up-to-date news, music, films, local events such as movie times or traffic conditions.

[7]`http://www.dbpedia.org`
[8]`http://www.dbis.informatik.uni-goettingen.de/Mondial/`
[9]`http://babelfish.yahoo.com/`

3.4.1 Aligning TPTP with SUMO and External Sources

Having a translated ACE text and external/internal sources that can provide world knowledge axioms related to that text is only useful if a WKR system can connect them. A TPTP query needs to be able to align with the available mediators otherwise, when the TPTP query is passed onto SPASS-XDB, it will not be able to make sense of the it. Since the mediators use a specific language to ask questions to the external sources, the TPTP query needs to include extra axioms that relate the things described in the mediator language to the things described in the query. Also, since SUMO is a source of world knowledge facts and relations, the tool must be able to align with it as well. These additional axioms act as the links between a the language construction of the query and the language of the world knowledge sources. The goal of the alignment tool, `get_Includes`, is to allow for ACE queries to be understood by SPASS-XDB so that it can perform the deduction process successfully.

First, since there are a finite number of mediators currently available, it is easy to allow for the conjecture to align with whichever one is necessary. The approach taken in the `get_Includes` tool to accomplish this is by scanning the TPTP conjecture for key words that relate to each mediator. For example, if the word 'weather' is found in a conjecture, `get_Includes` will add the necessary 'weather' related axioms; i.e., axioms that relate to all mediators pertaining to weather. This approach is successful, but by no means entirely efficient. The number of currently available mediators is small, but with the addition of a larger set, searching for key words would become a lengthy process or might even lead to issues concerning the overlap of some of those

words. Unfortunately, this tool is not fully automated yet, but is something to be considered for future research. The goal of this research is to create an end-to-end system, therefore worrying and preparing for possible future complications is beyond its scope.

Deciding which axioms are to be added when a certain key word is located in the conjecture is currently done by hand, in advance. These additional axioms are created beforehand and stored within `get_Includes` so they can be retrieved when needed. Example 3.11 shows the resulting axiom that is returned by `get_Includes` when the key word 'weather' is found. The mediator information (i.e., the information returned by the mediator) is the `with_weather` predicate and it is linked by an implication operator to four other predicates that indicate that the initial ACE query was related to the weather of a city (the prefix 'nd' is added by NACE to avoid clashes between similar predicate names in SUMO). The reason this axiom works is because it says that if a TPTP formula contains the listed predicates pertaining to the existence of a city and a weather condition, and the weather of that city is that very weather condition, then it is implied by the antecedent (the Weather mediator information specific to that city).

Example 3.11

```
weather,(![Country,USState,Condition,CityName,
Temp,Clouds,Humidity,Pressure,WindSpeed]:
(with_weather(Country,USState,Condition,CityName,
Temp,Clouds,Humidity,Pressure,WindSpeed)
        => (nd_weather_of(Condition,CityName)
              & nd_city_of(CityName,Country)
              & nd_city(CityName)
```

```
                        & nd_weather(Condition)))).
```

More examples are provided in the Chapter 4, whilst analyzing the tests and results.

3.5   Web Interface

To have a fully working WKR system in place, a web interface is needed to access the implemented system as a whole. The goal is to have a user-interface on the web that is simple and easy to use for any user. Not only is the web interface that was built for this research designed to query the CNL-WKR system, it is also designed to return different outputs based on what the user wants. Different flags are available in order to return output from each step of the logic translation, axiom retrieval, and/or proof.

The CNL-WKR system user-interface is written in Perl. The basic structure of the interface is outlined in Figure 3.2. The Perl script is made up of calls to other Prolog executables and services. First, the ACE text is sent off to the APE web service. This is done through the Prolog executable, get_DRS. It simply adds the ACE text to the tail-end of a URL which delivers (if a valid ACE text) the corresponding DRS as a Prolog term. This term is then passed through the next Prolog executable, get_FOF, where it is passed through NACE. NACE returns the corresponding TPTP first-order logic. Now, if it is known before hand that a conjecture will need some external source/SUMO alignment, a flag can be added to run get_FOF_SUMO instead of get_FOF. get_FOF_SUMO uses NACE as well, but it adds a prefix of 'nd' to each

logic predicate. This is done so terms do not clash with external sources or SUMO terms if they are coincidentally alike. Also, `get_FOF_SUMO` runs the translated and prefixed logic through `get_Includes` to retrieve the appropriate axioms. These are then added to the input string of logic. This logic string is then passed through TPTP4X [8] which transforms the input string into the properly pretty-printed TPTP form which SPASS-XDB requires. Finally, this form is passed onto SPASS-XDB where the reasoning process begins. If a proof is found, the results are returned to the user. Otherwise, the results explain that there was no proof found (which means the conjecture is not provable) or that the system gave up (which means the system does not have enough information to find a proof or non-proof).

It should also be noted that the user-interface can also accept a set of statements and a conjecture to see if the conjecture is a logical consequence of the set. External sources and/or SUMO do not necessarily have to be used. A simple flag can be attached so that this string of statements and the conjecture are passed through a separate Prolog executable, `get_Proof`. This executable, takes a string of ACE statements, converts all of them to logic using NACE (returning them as axioms except for any statement which begins with 'Therefore' - these are returned as conjectures). Since `get_DRS`, `get_FOF`, and `get_FOF_SUMO` all are designed on receiving one ACE text query, namely a conjecture, a separate executable had to be designed to accept multiple statements.
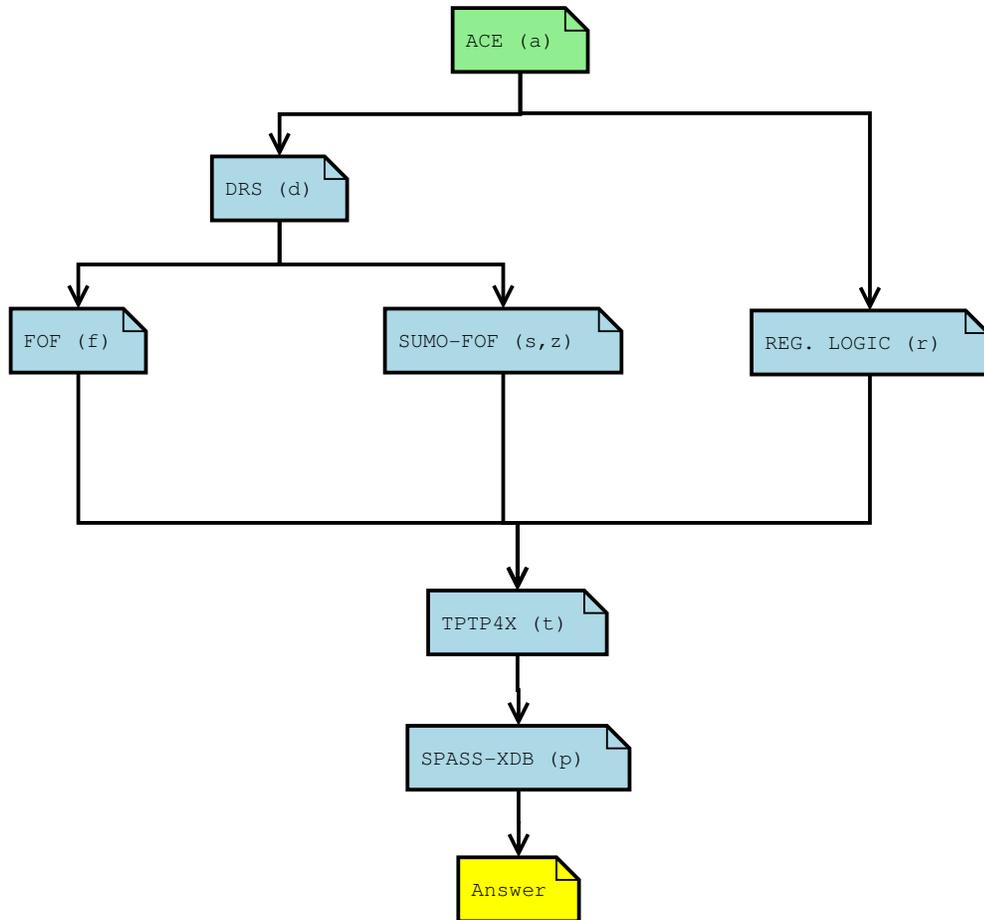
Figure 3.2: User-interface Process

Flags

Each of the steps described previously can be flagged. Combinations of flags can also be used. Here are the following flags and their functions:

1. a: returns ACE text

2. d: returns DRS retrieved from the ACE parser

3. f: returns TPTP via NACE

4. s: returns SUMO prefixed TPTP and extra axioms added through `get_Includes`

5. z: returns SUMO prefixed TPTP and adds extra axioms through `get_Includes` and includes the answer in a specifically printed line

6. r: runs regular logic proofs through `get_Proof`

7. t: returns pretty-printed TPTP4X

8. p: runs SPASS-XDB and returns proof output

9. x: exit

Note the difference between flag s and flag z. They both do the same thing except flag z adds a print line to the conjecture. This print line option allows the answer of a query to be specifically printed as 'Answer is X,' whereas flag s only allows for the system to print a proof.

Web Front-end

The web front-end was designed to make the **CNL-WKR** system more user-accessible and user-friendly. The flags described previously now are offered as checkboxes on a public website[10]:

The CNL-WKR System

☐ ACE
☐ DRS
☐ FOF
☐ SUMO-FOF
☐ SUMO-FOF w/ Printline
☐ Basic Logic
☐ TPTP4X Format
☐ Answer/Proof

```
Enter ACE here...
```

(Submit)

Figure 3.3: Web Front-end

3.6 Chapter 3 Summary

In this chapter, the algorithm for translating an ACE text into TPTP first-order logic was presented and explained. It was then shown how the translated text is then passed on to **SPASS**-**XDB** to be used as a conjecture. The description of the mediators that interface with external sources as well as the tool that aligns the conjecture with

---

[10]http://www.cs.miami.edu/~tptp/Guest/FOLconvert1.html

those sources, was also described in this chapter. Finally, the web interface and the script that lies beneath it was discussed.

Chapter 4

Testing and Results

This chapter discusses the testing performed on the CNL-WKR system and details the results. First, the testing of NACE, which completes the conversion from ACE to TPTP, is shown in Section 4.1. A large number of ACE-acceptable English sentences have been run through NACE and the results are shown here. The mediators that were created in this research are tested and the results are shown in Section 4.2. In Section 4.3, the entire CNL-WKR system is tested from end-to-end with examples that vary in degree of difficulty and that test different aspects of the system. In this section, the testing of `get_Includes` and its results are also documented, since the success of the WKR system as a whole is largely dependent on the alignment tools ability to add the correct additional axioms. Finally, Section 4.4 provides a brief summary of the chapter.

4.1   Testing NACE

The number of different allowable ACE sentences are infinitely many. In this section, a large selection of common ACE sentence constructs are provided. Each group demonstrates NACE's capability of translating different DRS elements. This is by no means a complete set of examples, for there are infinitely many possible ACE sentences. The examples in this section include the ACE text being tested, then the corresponding DRS, and then the final translated logic by NACE.

The only other tool that is currently available to translate ACE to TPTP first order logic is the one provided by the ACE tool. It was considered sub-par because it does not unwrap the DRS elements fully. Simple phrases are done well, but elements as simple as verbs are introduced, the structure begins to look messy like in Example 4.1. If you compare this result to Example 4.8 later on in this section, you will see that NACE completely unwraps the `predicate` predicate.

Example 4.1 *The man wins.*

```
fof(f1, axiom, (
    ? [A, B] :
      ( ( predicate1(A, win, B) )
        & ( man(B) ) ) )).
```

4.1.1   Simple Sentences

The following examples demonstrate NACE's capability of translating simple sentences that have one object and mention the existence of that one object (or plural object).

Example 4.2 *There is a man.*

```
drs([A],[object(A,man,countable,na,eq,1)-1/4])

fof(name1,conjecture,(
    ? [A] : man(A) )).
```

Example 4.3 *There are 2 men.*

```
drs([A],[object(A,man,countable,na,eq,2)-1/4])

fof(name1,conjecture,(
    ? [A,B] :
      ( man(A)
      & man(B)
      & look_different(A,B) ) )).
```

Example 4.4 *There are less than 3 men.*

```
drs([A],[object(A,man,countable,na,less,3)-1/6])

fof(name1,conjecture,(
    ! [A,B,C] :
      ( ( man(A)
        & man(B)
        & man(C) )
     => ( A = B
        | A = C
        | B = C ) ) )).
```

Example 4.5 *There is more than 1 man.*

```
drs([A],[object(A,man,countable,na,greater,1)-1/6])

fof(name1,conjecture,(
    ? [A,B] :
      ( man(A)
      & man(B)
      & look_different(A,B) ) )).
```

Example 4.6 *There is some food.*

```
drs([A],[object(A,food,mass,na,na,na)-1/4])

fof(name1,conjecture,(
    ? [A] :
      ( beer(A)
      & weight(A,kg) > 0 ) )).
```

Example 4.7 *There is someone.*

```
drs([A],[object(A,somebody,countable,na,eq,1)-1/4])

fof(name1,conjecture,(
    ? [A] : somebody(A) )).
```

4.1.2   Action Sentences

The following examples demonstrate NACE's capability of translating sentences that contain action words (verbs). These sentences range in complexity. Some have one action and one object, while more complex ones have plural objects and multiple actions.

Example 4.8 *The man wins.*

```
drs([A,B],[predicate(A,win,B)-1/3,
            object(B,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A] :
      ( man(A)
      & win(A) ) )).
```

Example 4.9 *The man wins and drinks.*

```
drs([A,B,C],[predicate(A,win,C)-1/3,predicate(B,drink,C)-1/5,
            object(C,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A] :
      ( drink(A)
      & man(A)
      & win(A) ) )).
```

Example 4.10 *The man wins a prize.*

```
drs([A,B,C],[object(A,prize,countable,na,eq,1)-1/5,
            predicate(B,win,C,A)-1/3,
```

```
        object(C,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A,B] :
      ( man(A)
      & prize(B)
      & win(A,B) ) )).
```

Example 4.11 *The man wins a prize and wins a beer.*

```
drs([A,B,C,D,E],[object(A,prize,countable,na,eq,1)-1/5,
        predicate(B,win,E,A)-1/3,
        object(C,beer,countable,na,eq,1)-1/9,
        predicate(D,win,E,C)-1/7,
        object(E,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A,B,C] :
      ( beer(A)
      & man(B)
      & prize(C)
      & win(B,C)
      & win(B,A) ) )).
```

Example 4.12 *2 men win and lose.*

```
drs([A,B,C],[object(A,man,countable,na,eq,2)-1/2,
        predicate(B,win,A)-1/3,predicate(C,lose,A)-1/5])

fof(name1,conjecture,(
    ? [A,B] :
      ( lose(A)
      & lose(B)
      & win(A)
      & win(B)
      & man(B)
      & man(A)
      & look_different(B,A) ) )).
```

Example 4.13 *The man wins 2 prizes.*

```
drs([A,B,C],[object(A,prize,countable,na,eq,2)-1/5,
        predicate(B,win,C,A)-1/3,
        object(C,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
```

```
    ? [A,B,C] :
      ( man(A)
      & prize(B)
      & prize(C)
      & look_different(B,C)
      & win(A,C)
      & win(A,B) ) )).
```

Example 4.14 *2 men win a prize.*

```
drs([A,B,C],[object(A,man,countable,na,eq,2)-1/2,
            object(B,prize,countable,na,eq,1)-1/5,
            predicate(C,win,A,B)-1/3])

fof(name1,conjecture,(
    ? [A,B,C] :
      ( prize(A)
      & man(B)
      & man(C)
      & look_different(B,C)
      & win(C,A)
      & win(B,A) ) )).
```

Example 4.15 *The man wins and wants a beer.*

```
drs([A,B,C,D],[predicate(A,win,D)-1/3,
            object(B,beer,countable,na,eq,1)-1/7,
            predicate(C,want,D,B)-1/5,
            object(D,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A,B] :
      ( beer(A)
      & man(B)
      & win(B)
      & want(B,A) ) )).
```

Example 4.16 *The man gives a cookie to a dog.*

```
drs([A,B,C,D],[object(A,cookie,countable,na,eq,1)-1/5,
            object(B,dog,countable,na,eq,1)-1/8,
            predicate(C,give,D,A,B)-1/3,
            object(D,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A,B,C] :
```

```
( cookie(A)
& dog(B)
& man(C)
& give(C,A)
& give_to(C,A,B) ) )).
```

4.1.3   Copula Sentences

The following examples demonstrate NACE's capability of translating sentences that
contain copular verbs, or linking verbs, like the verb 'to be' These are special verbs
as they can indicate group membership and also provide a link between the subject
of a sentence and an object or adjective. It is also valid to translate copular verbs
into verb predicates (similar to what is done for regular verbs) but it is more intuitive
to translate them by extracting the verb name and simply changing the reference
variable of the thing being linked to the subject, to that of the subject.

Example 4.17 *The man is a winner.*

```
drs([A,B,C],[object(A,winner,countable,na,eq,1)-1/5,
        predicate(B,be,C,A)-1/3,
        object(C,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A] :
    ( man(A)
    & winner(A) ) )).
```

Example 4.18 *The man is a winner and the woman is a loser.*

```
drs([A,B,C,D,E,F],[object(C,man,countable,na,eq,1)-1/2,
        object(A,winner,countable,na,eq,1)-1/5,
        predicate(B,be,C,A)-1/3,
        object(F,woman,countable,na,eq,1)-1/8,
        object(D,loser,countable,na,eq,1)-1/11,
        predicate(E,be,F,D)-1/9])
```

```
fof(name1,conjecture,(
    ? [A,B] :
      ( loser(A)
      & man(B)
      & winner(B)
      & woman(A) ) )).
```

Example 4.19 *The man is successful*

```
drs([A,B,C],[property(A,successful,pos)-1/4,
          predicate(B,be,C,A)-1/3,
          object(C,man,countable,na,eq,1)-1/2])
```

```
fof(name1,conjecture,(
    ? [A] :
      ( man(A)
      & successful(A) ) )).
```

Example 4.20 *The man is successful and confident.*

```
drs([A,B,C],[property(A,successful,pos)-1/4,
          property(A,confident,pos)-1/6,
          predicate(B,be,C,A)-1/3,
          object(C,man,countable,na,eq,1)-1/2])
```

```
fof(name1,conjecture,(
    ? [A] :
      ( man(A)
      & successful(A)
      & confident(A) ) )).
```

4.1.4 Prepositional Sentences

The following examples demonstrate NACE's capability of translating sentences that

contain prepositions.

Example 4.21 *The prize is for the man.*

```
drs([A,B,C],[object(C,prize,countable,na,eq,1)-1/2,
          object(B,man,countable,na,eq,1)-1/6,
          predicate(A,be,C)-1/3,modifier_pp(A,for,B)-1/4])
```

```
fof(name1,conjecture,(
    ? [A,B] :
      ( man(A)
      & prize(B)
      & is_for(B,A) ) )).
```

Example 4.22 *The man is with a dog.*

```
drs([A,B,C],[object(C,man,countable,na,eq,1)-1/2,
          object(A,dog,countable,na,eq,1)-1/6,
          predicate(B,be,C)-1/3,modifier_pp(B,with,A)-1/4])

fof(name1,conjecture,(
    ? [A,B] :
      ( dog(A)
      & man(B)
      & is_with(B,A) ) )).
```

Example 4.23 *The man wins in the bank.*

```
drs([A,B,C],[object(C,man,countable,na,eq,1)-1/2,
          object(B,bank,countable,na,eq,1)-1/6,
          predicate(A,win,C)-1/3, modifier_pp(A,in,B)-1/4])

fof(name1,conjecture,(
    ? [A,B] :
      ( bank(A)
      & man(B)
      & win(B)
      & win_in(B,A) ) )).
```

Example 4.24 *The man wins a prize for a team.*

```
drs([A,B,C,D],[object(A,team,countable,na,eq,1)-1/8,
          object(B,prize,countable,na,eq,1)-1/5,
          predicate(C,win,D,B)-1/3,modifier_pp(C,for,A)-1/6,
          object(D,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
    ? [A,B,C] :
      ( team(A)
      & man(B)
      & prize(C)
      & win(B,C)
      & win_for(B,C,A) ) )).
```

4.1.5 Adverb Sentences

The following examples demonstrate NACE's capability of translating sentences that contain adverbs.

Example 4.25 *The man wins quickly.*

```
drs([A,B],[object(B,man,countable,na,eq,1)-1/2,
           predicate(A,win,B)-1/3,
           modifier_adv(A,quickly,pos)-1/4])

fof(name1,conjecture,(
    ? [A] :
      ( man(A)
      & win(A)
      & win_quickly(A) ) )).
```

Example 4.26 *The man wins quickest.*

```
drs([A,B],[object(B,man,countable,na,eq,1)-1/2,
           predicate(A,win,B)-1/3,
           modifier_adv(A,quick,sup)-1/4])

fof(name1,conjecture,(
    ? [A] :
      ( man(A)
      & win(A)
      & win_most_quick(A) ) )).
```

Example 4.27 *The man wins a prize quickly.*

```
drs([A,B,C],[object(C,man,countable,na,eq,1)-1/2,
             object(A,prize,countable,na,eq,1)-1/5,
             predicate(B,win,C,A)-1/3,
             modifier_adv(B,quickly,pos)-1/6])

fof(name1,conjecture,(
    ? [A,B] :
      ( man(A)
      & prize(B)
      & win(A,B)
      & win_quickly(A,B) ) )).
```

Example 4.28 *The man wins a prize more quickly.*

```
drs([A,B,C],[object(C,man,countable,na,eq,1)-1/2,
             object(A,Prize,countable,na,eq,1)-1/5,
             predicate(B,win,C,A)-1/3,
             modifier_adv(B,quickly,comp)-1/7])

fof(name1,conjecture,(
    ? [A,B] :
      ( man(A)
      & prize(B)
      & win(A,B)
      & win_more_quickly(A,B) ) )).
```

4.1.6 Relational Sentences

The following examples demonstrate NACE's capability of translating sentences that contain the relational preposition, of. ACE considers of to be a special preposition since it is typical used to link the subject to another object (much like a copular verb).

Example 4.29 *A man is a member of a family.*

```
drs([A,B,C,D],[object(A,man,countable,na,eq,1)-1/2,
               object(B,member,countable,na,eq,1)-1/5,
               object(C,family,countable,na,eq,1)-1/8,
               relation(B,of,C)-1/6,predicate(D,be,A,B)-1/3])

fof(name1,conjecture,(
    ? [A,B] :
      ( family(A)
      & man(B)
      & member(B)
      & member_of(B,A) ) )).
```

Example 4.30 *The winner of a prize is a man.*

```
drs([A,B,C,D],[relation(C,of,D)-1/3,
               object(D,prize,countable,na,eq,1)-1/5,
```

```
                        object(C,winner,countable,na,eq,1)-1/2,
                        object(A,man,countable,na,eq,1)-1/8,
                        predicate(B,be,C,A)-1/6])

        fof(name1,conjecture,(
            ? [A,B] :
              ( man(A)
              & prize(B)
              & winner(A)
              & winner_of(A,B) ) )).
```

Example 4.31 *The winner of a prize wins.*

```
        drs([A,B,C],[relation(B,of,C)-1/3,
                     object(C,prize,countable,na,eq,1)-1/5,
                     object(B,winner,countable,na,eq,1)-1/2,
                     predicate(A,win,B)-1/6])

        fof(name1,conjecture,(
            ? [A,B] :
              ( prize(A)
              & win(B)
              & winner(B)
              & winner_of(B,A) ) )).
```

4.1.7   Conditional Sentences

The following examples demonstrate NACE's capability of translating conditional statements.

Example 4.32 *Every man wins a prize.*

```
        drs([],[=>(drs([A],[object(A,man,countable,na,eq,1)-1/2]),
                   drs([B,C],[object(B),prize,countable,na,eq,1)-1/5,
                   predicate(C,win,A,B)-1/3]))])

        fof(name1,conjecture,(
            ! [A] :
              ( man(A)
            => ? [B] :
                 ( prize(B)
                 & win(A,B) ) ) )).
```

Example 4.33 *Every man succeeds.*

```
drs([],[=>(drs([A],[object(A,man,countable,na,eq,1)-1/2]),
          drs([B],[predicate(B,succeed,A)-1/3]))])

fof(name1,conjecture,(
    ! [A] :
      ( man(A)
    => succeed(A) ) )).
```

Example 4.34 *Every man gives a cookie to a dog.*

```
drs([],[=>(drs([A],[object(A,man,countable,na,eq,1)-1/2]),
          drs([B,C,D],[object(B,cookie,countable,na,eq,1)-1/5,
          object(C,dog,countable,na,eq,1)-1/8,
          predicate(D,give,A,B,C)-1/3]))])

fof(name1,conjecture,(
    ! [A] :
      ( man(A)
    => ? [B,C] :
          ( dog(B)
          & cookie(C)
          & give(A,C)
          & give_to(A,C,B) ) ) )).
```

4.1.8   Disjunction Sentences

The following examples demonstrate NACE's capability of translating sentences that
contain disjunctions.

Example 4.35 *There is a man or there is a dog.*

```
drs([],[v(drs([A],[object(A,man,countable,na,eq,1)-1/4]),
          drs([B],[object(B,dog,countable,na,eq,1)-1/9]))])

fof(name1,conjecture,
    ( ? [A] : man(A)
    | ? [B] : dog(B) )).
```

Example 4.36 *Every man wins or the dog barks.*

```
drs([],[v(drs([],[=>(drs([A],
          [object(A,man,countable,na,eq,1)-1/2]),
          drs([B],[predicate(B,win,A)-1/3]))]),
          drs([C,D],[predicate(C,bark,D)-1/9]))])

fof(name1,conjecture,
    ( ! [A] :
        ( man(A)
      => ? [B] :
            ( prize(B)
            & win(A,B) ) )
    | ? [C] :
        ( dog(C)
        & bark(C) ) )).
```

4.1.9 Sentences Involving Names

The following examples demonstrate NACE's capability of translating sentences that contain defined variables (i.e., names of things). Names must be treated differently because they need to be considered separately from variables, so that they do not get quantified.

Example 4.37 *Curie wins a prize.*

```
drs([A,B],[object(A,prize,countable,na,eq,1)-1/4,
          predicate(B,win,named('Marie-Curie'),A)-1/2])

fof(name1,conjecture,(
    ? [A] :
      ( prize(A)
      & win('Marie-Curie',A) ) )).
```

Example 4.38 *The man gives a beer to Marie-Curie.*

```
drs([A,B,C],[object(A,beer,countable,na,eq,1)-1/5,
          predicate(B,give,C,A,named('Marie-Curie'))-1/3,
          object(C,man,countable,na,eq,1)-1/2])

fof(name1,conjecture,(
```

```
    ? [A,B] :
      ( beer(A)
      & man(B)
      & give(B,A)
      & give_to(B,A,'Marie-Curie') ) )).
```

Example 4.39 *Albert-Einstein sings a song with Marie-Curie.*

```
    drs([A,B],[object(A,song,countable,na,eq,1)-1/4,
              predicate(B,sing,named('Albert-Einstein'),A)-1/2,
              modifier_pp(B,with,named('Marie-Curie'))-1/5])

    fof(name1,conjecture,(
        ? [A] :
          ( song(A)
          & sing('Albert-Einstein',A)
          & sing_with('Albert-Einstein',A,'Marie-Curie') ) )).
```

Example 4.40 *Geoff is a member of a family.*

```
    drs([A,B,C],[object(A,member,countable,na,eq,1)-1/4,
                object(B,family,countable,na,eq,1)-1/7,
                relation(A,of,B)-1/5,
                predicate(C,be,named('Geoff'),A)-1/2])

    fof(name1,conjecture,(
        ? [A] :
          ( family(A)
          & member('Geoff')
          & member_of('Geoff',A) ) )).
```

4.2   Testing Mediators

The mediators for SPASS-XDB provide static and dynamic world knowledge axioms.

Even though SPASS-XDB currently uses only a handful of mediators, the ones that

have been implemented thus far provide some interesting and encouraging results.

The mediators that were designed for this research are the following:

1. XChange Mediator

2. Location Mediator

3. Weather Mediator

4. With_weather Mediator

5. AmazonBooks Mediator

The results of their usage is demonstrated here. Their testing was done using the SystemQATPTP website[1], developed by Geoff Sutcliffe [26]. The website has an interface where the external sources of axioms are directly accessible, without the use of a WKR system. The user specifies a query in the SPASS-XDB-friendly format, which is then sent off to the external source via mediator, and the resulting answers are delivered.

4.2.1 XChange Mediator

The format with which to pose a query to the XChange mediator is the following:

```
fof(!Name,question,? [ToAmount] :
            xchange(!FromAmount,!FromCode,ToAmount,!ToCode) ).
```

In this form, all variables with exclamation marks are variables that need to be instantiated. `Name` is the name of the query, `FromAmount` is the currency amount one is trying to convert, `FromCode` is the code of the currency being translated, `ToAmount` is the converted currency amount, and `ToCode` is the converted currency code. Caution should be taken when entering currency codes in capital letters because Prolog will

---

[1] www.tptp.org/cgi-bin/SystemQATPTP

assume these to be variables. The currency code should be written in lower case or written in single quotes.

Example 4.41 *Query:*

```
fof(a,question,?[ToAmount]:xchange(5.00,'USD',ToAmount,'EUR')).
```

Answer:

```
fof(xchange,answer,xchange(5,'EUR',6.19,'USD'),answer_to(a,[])).
```

Example 4.42 *Query:*

```
fof(a,question,?[ToAmount]:xchange(0.03,'GBP',ToAmount,'RUB')).
```

Answer:

```
fof(xchange,answer,xchange(0.03,'GBP',1.34,'RUB'),
          answer_to(a,[])).
```

4.2.2 Location Mediator

The format with which to pose a query to the Location mediator is the following:

```
fof(!Name,question,?[LocationHints,CityName,CountryName,
          Latitude,Longitude,Elevation]:location(!Location,
          LocationHints,CityName,CountryName,Latitude,
          Longitude,Elevation)).
```

In this form, `Location` is the place that is being queried about. This place can be a city or even a landmark or general area. `LocationHints` is an optional variable which can be instantiated to the country name of the place being queried. There might be numerous cities with the same world, so specifying the country would narrow the results. `CityName` and `CountryName` are the city and country that is found and returned. `Latitude`, `Longitude`, and `Elevation` are the latitude, longitude, and elevation of the place being queried.

Example 4.43 *Query:*

```
fof(a,question,?[LocationHints,CityName,CountryName,Latitude,
          Longitude,Elevation]:location('Paris','France',
          CityName,CountryName,Latitude,Longitude,Elevation)).
```

Answer:

```
fof(location,answer,location('Paris','France','Paris','France',
          48.85341,2.3488,30),answer_to(a,[])).
```

Example 4.44 *Query:*

```
fof(a,question,? [LocationHints,CityName,CountryName,Latitude,
          Longitude,Elevation]:location('Mt. Everest',
          LocationHints,CityName,CountryName,Latitude,Longitude,
          Elevation)).
```

Answer:

```
fof(location,answer,location('Mt. Everest','Nepal','Peak XV',
          'Nepal',27.989096,86.924644, 8554), answer_to(a,[])).
```

In Example 4.43, the query is for Paris, France. The provided answer is actually only the first answer returned of six. Each of those six were Paris, France, but indicated a different area of the city. In the case of Example 4.44, Mt. Everest was queried and only one exact match was returned.

4.2.3 Weather Mediator

The format with which to pose a query to the Weather mediator is the following:

```
fof(!Name,question,?[LocationHints,CityName,CountryName,
          Temperature,Clouds,Precipitation,Humidity,Pressure,
          Windspeed]:weather(Location,LocationHints,CityName,
          CountryName,Temperature,Clouds,Precipitation,
          Humidity,Pressure,Windspeed)).
```

The variables in the previous query format are similar to those for the Location mediator query format except that instead of returning information about the latitude, longitude, and elevation of a location, the mediator returns facts about weather: temperature (in Celsius), cloud cover, precipitation, humidity (in percentage), pressure (in millibars), and windspeed (in mph). This mediator is an extension of the Location mediator because it shares the same weather source, but provides a different domain of information.

Example 4.45 *Query:*

```
fof(a,question,?[LocationHints,CityName,CountryName,Temperature,
          Clouds,Precipitation,Humidity,Pressure,Windspeed]:
          weather('Paris','France',CityName,CountryName,
          Temperature,Clouds,Precipitation,Humidity,Pressure,
          Windspeed)).
```

Answer:

```
fof(weather,answer,weather('Paris','France','Paris','France',17,
          'scattered clouds','n/a',59,1015.0,14),answer_to(a,[])).
```

Example 4.46 *Query:*

```
fof(a,question,?[LocationHints,CityName,CountryName,Temperature,
          Clouds,Precipitation,Humidity,Pressure,Windspeed]:
          weather('Kathmandu',LocationHints,CityName,
          CountryName,Temperature,Clouds,Precipitation,
          Humidity,Pressure,Windspeed)).
```

Answer:

```
fof(weather,answer,weather('Kathmandu','Nepal','Kathmandu',
          'Nepal',28,'few clouds','n/a',16,1010.0,5),
          answer_to(a,[])).
```

It should be noted that there are only weather forecasts available in areas where there are accessible weather stations. For example, entering Mt. Everest as a location does not provide any weather information since it is at a location where weather stations are not publicly accessible.

4.2.4   With_Weather Mediator

The format with which to pose a query to the With_weather mediator is the following:

```
fof(!Name,question,?[Country,USAState,WeatherCondition,CityName,
        Temperature,Clouds,Humidity,Pressure,Windspeed]:
        with_weather(!Country,!USAState,WeatherCondition,
        CityName,Temperature,Clouds,Humidity,Pressure,
        Windspeed)).
```

In this form, the `Country` is the country being queried about. If the country is USA, then a state must be specific in `USAState`, otherwise this is left as an uninstantiated variable. `WeatherCondition` is the desired weather condition to be searched for. Possible values for this variable are limited to roughly 30 different conditions defined by Yahoo Weather. The predefined conditions are based on the most common phrases used in daily weather forecasts such as *partly cloudy, sunny, snow, light showers*, etc. These `WeatherCondition` inputs can be a partial string match. Places that are partly cloudy are also cloudy, so the mediator will match the input of `cloudy` to '`partly cloudy`' or '`mostly cloudy`', for example. Once the condition is matched with a city in the specified country, other weather facts are appended, similar to the ones in the Weather mediator (except for temperature, which is given in Fahrenheit).

Example 4.47 *Query:*

```
fof(a,question,?[Country,USAState,WeatherCondition,CityName,
        Temperature,Clouds,Humidity,Pressure,Windspeed]:
        with_weather(france,USAState,'Cloudy',CityName,
        Temperature,Clouds,Humidity,Pressure,Windspeed)).
```

Answer:

```
fof(weather,answer,with_weather(france,non_usa,'Cloudy',
         'Auris En Oisans',73,'Mostly Cloudy',47,30.03,5),
         answer_to(a,[])).
```

Example 4.48 *Query:*

```
fof(a,question,?[Country,USAState,WeatherCondition,CityName,
         Temperature,Clouds,Humidity,Pressure,Windspeed]:
         with_weather(usa,florida,'Sunny',CityName,Temperature,
         Clouds,Humidity,Pressure,Windspeed)).
```

Answer:

```
fof(weather,answer,with_weather(usa,florida,'Sunny',
         'Delray Beach',88,'Sunny',62,29.85,9),
         answer_to(a,[])).
```

The answers provide in Example 4.47 and Example 4.48 were the first of many to
be returned. Obviously, there may be cities in a country that share the same weather
condition.

4.2.5 AmazonBooks Mediator

The format with which to pose a query to the AmazonBooks mediator is the following:

```
fof(!Name,question,?[AuthorName,Title,Binding,Date,Price,Currency]:
         book(!Author,AuthorName,Title,Binding,Date,Price,
         Currency)).
```

In this form, the `Author` variable is the name of an author. The spelling and structure of the author name can be queried loosely since the mediator uses Amazon's search capabilities. `AuthorName` is the returned full name of the author. `Title` is the title of a book written by that author. `Binding` is the type of binding of the book (paperback or hardcover). `Date` is the release date of the book. `Price` and `Currency` indicate the price and currency of the book. Multiple answers are returned, but only the first is shown here. Amazon can do order the results in different ways, but the default ordering is done by most popular to least popular.

Example 4.49 *Query:*

```
fof(a,question,?[AuthorName,Title,Binding,Date,Price,Currency]:
        book('Douglas Hofstadter',AuthorName,Title,Binding,
        Date,Price,Currency)).
```

Answer:

```
fof(book,answer,book('Douglas Hofstadter','Douglas R.
        Hofstadter','Godel, Escher, Bach: An Eternal Golden
        Braid','Paperback','1999-02-05',22.95,'USD'),
        answer_to(a,[])).
```

4.3   Testing the Full CNL-WKR System

Here, the testing of the full CNL-WKR system is documented, by using the Perl script interface that combines all the different tools (discussed in the previous chapter). In the previous sections, the mediators and the logic translator were tested and the

results were shown. The success of the CNL-WKR system relies highly on the ability of the `get_Includes` tool to work properly, so the capabilities of `get_Includes` are demonstrated here. The testing done in the next few sections on the following types of examples covers four different levels of complexity:

1. Non-XDB (basic proofs)

2. Simple SPASS-XDB with mediators and Alignment

3. Complex SPASS-XDB with mediators and Alignment

4. Full SPASS-XDB with mediators and SUMO

The examples in this section first gives the ACE query being tested, then gives the resulting pretty-printed TPTP translation, and then finally the result of the SPASS-XDB output.

At the time this research was conducted, there was no other known system that was capable of WKR. For this reason, there has been no comparative study made here between the CNL-WKR system and other WKR systems. Only recently has True Knowledge[2] released their public online beta version which is capable of WKR. In the future, to compare the two systems, the query examples below (which are quite complex) should be submitted to the True Knowledge system in order to show that although True Knowledge is capable of reasoning and answering basic queries, it can not answer the complex queries provided in this section. Because the CNL-WKR system uses external sources to retrieve world knowledge, it could potentially use the

---

[2]`http://www.trueknowledge.com`

True Knowledge API in the future. Using their API would allow for the CNL-WKR system to be able to answer all the questions True Knowledge is capable of answering, in addition to the complex queries demonstrated in this section.

4.3.1 Non-XDB

Here, two non-XDB examples are run through SPASS-XDB. By non-XDB, it is meant that the examples do not require internal/external axioms. Examples like these themselves are a complete set of logical premises with a conjecture to be proven. These kind of problems need to be flagged with the *-r* flag so that they can be run through `get_Proof`, which deals with queries that are greater than one sentence (i.e., not a sole query). The resulting TPTP is then passed on to SPASS-XDB where either a proof or no proof is found.

Example 4.50 *All men are mortal. Socrates is a man. Therefore, Socrates is mortal.*

```
fof(name1,axiom,(
    ! [A] :
      ( man(A)
    => mortal(A) ) )).

fof(name2,axiom,(
    man('Socrates') )).

fof(name3,conjecture,(
    mortal('Socrates') )).
```

   RESULT: Theorem Found

Example 4.51 *All dogs bark. Nelson does not bark. Therefore, Nelson is not a dog.*

```
fof(name1,axiom,(
    ! [A] :
      ( dog(A)
   => bark(A) ) )).

fof(name2,axiom,(
    ~ bark('Nelson') )).

fof(name3,conjecture,(
    ~ dog('Nelson') )).
```

RESULT: Theorem Found

4.3.2   Simple SPASS-XDB with Mediators and Alignment

Here are four examples that demonstrate the successful usage of the mediators and the alignment tool to answer queries given in ACE about specific domains of interest. The alignment axioms are added by `get_Includes`. To make the added axioms clear, they have been named `alignment1`, `alignment2`, etc. The range of queries relate only to the mediators that were created for this research. No example is provided for the XChange mediator because its syntax does not currently allow for very coherent queries. Here the flag for choosing a direct answer as the output (*-z*) was chosen.

Example 4.52 *There is a book that Larry-Wos writes.*

```
fof(name1,conjecture,(
    ? [A] :
      ( nd_book(A)
      & nd_write('Larry Wos',A)
      & print(printall(nl,'Answer is ',A,nl)) ) )).

fof(alignment1,axiom,(
    ! [Author,AuthorName,Title,Binding,Date,Price,Currency] :
```

```
      ( book(Author,AuthorName,Title,Binding,Date,Price,Currency)
   => ( nd_book(Title)
      & nd_write(Author,Title) ) ) )).
```

RESULT: 'Answer is 'Automated Reasoning: 33 Basic Research Problems''

Example 4.53 *There is a city of France and the weather of the city is 'Cloudy'.*

```
   fof(name1,conjecture,(
      ? [A] :
        ( nd_city(A)
        & nd_weather('Cloudy')
        & nd_city_of(A,'France')
        & nd_weather_of('Cloudy',A)
        & print(printall(nl,'Answer is ',A,nl)) ) )).

   fof(alignment1,axiom,(
      ! [B,C,D,E,F,G,H,I,J] :
        ( with_weather(B,C,D,E,F,G,H,I,J)
      => ( weather_of(D,E)
         & nd_city_of(E,B)
         & nd_city(E)
         & nd_weather(D) ) ) )).
```

RESULT: 'Answer is 'Font Romeu''

Example 4.54 *There is a city and the city is Paris and the city has an elevation.*

```
   fof(name1,conjecture,(
      ? [A] :
        ( nd_city('Paris')
        & nd_elevation(A)
        & nd_have('Paris',A)
        & print(printall(nl,'Answer is ',A,nl)) ) )).

   fof(name2,axiom,(
      ! [A,B,C,D,E,F,G] :
        ( location(A,B,C,D,E,F,G)
```

```
=> ( nd_city(A)
   & nd_elevation(G)
   & nd_have(A,G) ) ) )).
```

RESULT: 'Answer is 30'

Example 4.55 *There is a city and the city is London and the city has a windspeed.*

```
fof(name1,conjecture,(
    ? [A] :
      ( nd_city('London')
      & nd_windspeed(A)
      & nd_have('London',A)
      & print(printall(nl,'Answer is ',A,nl)) ) )).

fof(alignment1,axiom,(
    ! [A,B,C,D,E,F,G,H,I,J] :
      ( weather(A,B,C,D,E,F,G,H,I,J)
    => ( nd_city(A)
       & nd_windspeed(J)
       & nd_have(A,J) ) ) )).
```

RESULT: 'Answer is 6'

The results here are encouraging as they demonstrate the capability of the system to retrieve useful information about the world. Queries like this are quick and take a fraction of a second to return results (except for the With_weather mediator related queries, since they require some string matching).

4.3.3 Complex SPASS-XDB with Mediators and Alignment

Here are two examples that demonstrate SPASS-XDB's capabilities when using SUMO axioms, along with mediators and the alignment tool. Here, the capability of the system to use its internal source (SUMO) of knowledge is demonstrated.

Prize Problem

In this example, a question about a winner of two prizes and the name of a particular member of a specified family is queried for. SUMO has a large corpus of knowledge about prizes won by famous individuals, ranging from Peace Prizes to Nobel Prizes. The query is asked as follows (keep in mind that *family_member* requires an '*n*' preceding it because it is a noun that is not recognized by ACE):

Example 4.56 *There is a n:family_member of Curie and the n:family_member wins 2 prizes.*

```
fof(name1,conjecture,(
    ? [A,B,C] :
      ( nd_family_member(A)
      & nd_prize(B)
      & nd_prize(C)
      & look_different(B,C)
      & nd_family_member_of(A,'Curie')
      & nd_win(A,C)
      & nd_win(A,B)
      & print(printall(nl,'Answer is ',A,nl)) ) )).

fof(alignment1,axiom,(
    ! [A,B] :
      ( has_WonPrize(A,B)
    => ( nd_win(A,B)
       & nd_prize(B) ) ) )).

fof(alignment2,axiom,(
    ! [C,D,E] :
      ( ( is_familyName(D,E)
        & is_givenName(C,E) )
    => ( nd_family_member(E)
       & nd_family_member_of(E,D) ) ) )).
```

RESULT: 'Answer is 'Marie Curie''

Composer Problem

In this example, a question about a composer being born between two specified years is queried for. SUMO has a large corpus of knowledge about composers and their dates of birth. The query is posed in the following manner:

Example 4.57 *There is a composer that has a n:birthdate and the n:birthdate is after 1700 and the n:birthdate is before 1750.*

```
fof(name1,conjecture,(
    ? [A,B] :
      ( nd_birthdate(A)
      & nd_composer(B)
      & nd_have(B,A)
      & nd_is_after(A,int(1700))
      & nd_is_before(A,int(1750))
      & print(printall(nl,'Answer is ',B,nl)) ) )).

fof(alignment1,axiom,(
    ! [C] :
      ( is_instance(C,'Composer')
     => nd_composer(C) ) )).

fof(alignment2,axiom,(
    ! [D,E,F,G] :
      ( is_birthdate(D,E,F,G)
     => ( nd_birthdate(G)
        & nd_have(D,G) ) ) )).

fof(alignment3,axiom,(
    ! [H,I] :
      ( $lesseq(H,I)
     => nd_is_before(H,int(I)) ) )).

fof(alignment4,axiom,(
    ! [J,K] :
      ( $greatereq(J,K)
     => nd_is_after(J,int(K)) ) )).
```

RESULT: 'Answer is 'Carl Philipp Emanuel Bach''

The queries in the previous examples are quite complex and they are queries that most other comparative systems may not be able to answer.

4.3.4 Full SPASS-XDB with Mediators and SUMO

Finally, here are two examples that demonstrate the usage of multiple mediators and SUMO axioms. The mediators being used in the following examples are the LatLong mediator, which retrieves the latitude and longitude of a city, and the YAGOSUMO mediator, which interacts with SUMO. Additional SUMO axioms (that are added by hand) are used to provide information about cities that are prone to flooding. The added SUMO axioms include 'SUMO' in their axiom names. These examples show the true potential of a WKR system, where it can take external world knowledge and combine it with internal world knowledge to infer something that is interesting and useful.

The following query asks the system to prove that Abraham Lincoln is a mammal.

Example 4.58 *Abraham_Lincoln is a mammal.*

```
fof(name1,conjecture,(
    mammal('Abraham_Lincoln') )).

fof(alignment1,axiom,
    ( s__instance(s__AbrahamLincoln,s__Mammal)
  => mammal('Abraham_Lincoln') )).

fof(kb_SUMO_28,axiom,(
    ! [V__X,V__Y,V__Z] :
```

```
      ( ( s__instance(V__Y,s__SetOrClass)
        & s__instance(V__X,s__SetOrClass) )
     => ( ( s__subclass(V__X,V__Y)
          & s__instance(V__Z,V__X) )
       => s__instance(V__Z,V__Y) ) ) )).

fof(kb_SUMO_5811,axiom,(
    s__instance(s__Mammal,s__SetOrClass) )).

fof(kb_SUMO_5813,axiom,(
    s__subclass(s__Primate,s__Mammal) )).

fof(kb_SUMO_5818,axiom,(
    s__instance(s__Primate,s__SetOrClass) )).

fof(kb_SUMO_5823,axiom,(
    s__subclass(s__Hominid,s__Primate) )).

fof(kb_SUMO_5824,axiom,(
    s__instance(s__Hominid,s__SetOrClass) )).

fof(kb_SUMO_5826,axiom,(
    s__subclass(s__Human,s__Hominid) )).

fof(kb_SUMO_5836,axiom,(
    s__instance(s__Human,s__SetOrClass) )).
```

RESULT: Theorem found.

The following query asks the system for the name of a capital city that is prone to flooding and that shares the same (to the nearest degree) latitude as Moscow.

Example 4.59 *There is a capital city of an n:oecd_country and Moscow and the city have a latitude and the city v:floods.*

```
fof(name1,conjecture,(
    ? [A,B,C] :
      ( nd_capital(A)
```

```
      & nd_city(A)
      & nd_floods(A)
      & nd_latitude(B)
      & nd_oecd_country(C)
      & nd_city_of(A,C)
      & nd_have(A,B)
      & nd_have('Moscow',B) ) )).

fof(oecdcountry,axiom,(
    ! [Country,City,EnglishCity] :
      ( ( s__instance(Country,s__OECDMemberEconomiesClass)
        & s__capitalCity(City,Country)
        & xdb_ut(City,yagosumo,yagosumo,EnglishCity,english) )
     => ( nd_capital(City)
        & nd_city(City)
        & nd_city_of(City,Country)
        & nd_oecd_country(Country) ) ) )).

fof(moscow,axiom,(
    ! [EnglishCity,CityLat,CityLong,CityName,CityCountry,RoundLat,
    MoscowLat,MoscowLong,MoscowName,MoscowCountry] :
      ( ( look_different(EnglishCity,'Moscow')
        & latlong(EnglishCity,CityLat,CityLong,CityName,CityCountry)
        & $eval_real($round_real(CityLat),RoundLat)
        & latlong('Moscow',MoscowLat,MoscowLong,MoscowName,
        MoscowCountry)
        & $eval_real($round_real(MoscowLat),RoundLat) )
     => ( nd_have('Moscow',MoscowLat)
        & nd_latitude(MoscowLat)
        & nd_have(EnglishCity,CityLat)
        & nd_latitude(CityLat) ) ) )).

fof(floods,axiom,(
    ! [City] :
      ( s__capability(s__Flooding,s__located__m,City)
     => ( nd_floods(City)
        & nd_city(City) ) ) )).

ffof(kb_SUMO_28,axiom,(
    ! [O,P,Q] :
      ( ( s__instance(P,s__SetOrClass)
        & s__instance(O,s__SetOrClass) )
     => ( ( s__subclass(O,P)
          & s__instance(Q,O) )
       => s__instance(Q,P) ) ) )).
```

```
fof(kb_SUMO_MILO_6297,axiom,(
    s__instance(s__WaterArea,s__SetOrClass) )).

fof(kb_SUMO_MILO_10029,axiom,(
    s__instance(s__BodyOfWater,s__SetOrClass) )).

fof(kb_SUMO_MILO_DOMAINS_9645,axiom,(
    s__subclass(s__Sea,s__BodyOfWater) )).

fof(kb_SUMO_MILO_DOMAINS_9546,axiom,(
    s__subclass(s__BodyOfWater,s__WaterArea) )).

fof(kb_SUMO_MILO_DOMAINS_80407,axiom,(
    s__instance(s__Sea,s__SetOrClass) )).

fof(flood_near_water,axiom,(
    ! [R,S] :
      ( ( s__orientation(S,R,s__Near)
        & s__instance(R,s__WaterArea) )
     => s__capability(s__Flooding,s__located__m,S) ) )).

fof(coastal_cities_near_water,axiom,(
    ! [T] :
      ( s__instance_ground(T,s__CoastalCitiesClass)
     => ? [U] :
          ( s__instance(U,s__Sea)
          & s__orientation(T,U,s__Near) ) ) )).
```

RESULT: 'Answer is 'Copenhagen''

Even though a bit of human interaction was needed to direct the theorem prover towards this result, it is a positive indication for future results. The automation of adding additional axioms is something to be considered for future work.

4.4   Chapter 4 Summary

In this chapter, test results were documented and the full range of SPASS-XDB's capabilities were shown. First, the NACE logic translator was looked at and a large amount of ACE texts and their corresponding translated logic was shown. Next, the results from mediator usage was documented. Here, five of the more recently developed (specifically for this research) mediators were tested vigorously with promising results. The SUMO/external source alignment tool was also tested and the resulting delivered axioms were shown. Finally, the full CNL-WKR system was tested at four different levels of complexity. Each test provided a different view into the capabilities and potential of the whole system.

Chapter 5

Conclusion

This final chapter summarizes the entire thesis and provides a breakdown of what goals were achieved and what goals were not. Section 5.1 provides a review of the entire thesis and highlights the key achievements. Section 5.2 discusses the goals that were achieved and the contributions that this thesis has made to the science community. Finally, Section 5.3 mentions all the ideas that did not work out as initially planned, as well as aspects of this research that were out of scope. Also, ideas for future work in this area of research are proposed.

5.1 Thesis Review

This thesis has given a detailed description of what a WKR system is, what pieces are needed to build one, and what one can be capable of. It was shown that the pieces needed to build a WKR system are a user-friendly interface, sources of world knowledge (either external or internal), and a reasoning engine. The CNL-WKR system,

which was the focus of this research, provides all three pieces. This was demonstrated over the four previous chapters.

In Chapter 1, an explanation of world knowledge and WKR was given, followed by a brief review of the history of WKR systems. Then, an introduction to the CNL-WKR system was given. In Chapter 2, a background about CNLs, NLIs, and WKR systems was given. In this chapter, the ACE language was described in detail, since it was the query language of choice for the CNL-WKR system. Chapter 3 discussed the CNL-WKR system architecture design and how it was implemented. Details on the SPASS-XDB theorem prover, which the CNL-WKR system uses, were also given here. Chapter 4 showed the results of exhaustive testing on each part of the CNL-WKR system. The first testing was done on NACE, which completes the translation from ACE to TPTP. Next, each of the developed mediators (for this research) were tested and their capabilities demonstrated. Then, the full CNL-WKR system was tested at four different levels of complexity, ranging from proving basic logical consequence (without need of external/internal sources) from a set of ACE sentences, to finding answers to complex queries that require multiple sources to be used. Finally, in Chapter 5, the goals and contributions, as well as possible future work, were discussed.

5.2   Goals Achieved and Contributions

There were several goals that were aimed for in this research. The main goal of this research was to provide an end-to-end WKR system, where a user-query can be submitted at one end, and via WKR, a proof or answer can be returned at the other.

This goal was successfully accomplished by the CNL-WKR system.

One of the lesser important goals was to create a tool that successfully translated ACE queries into TPTP first-order logic. The tool that was created to do this was NACE. NACE proved to be a reliable and successful tool, as could be seen throughout the Chapter 4 testing. Another lesser goal was to provide new and interesting external sources that covered new and previously unexplored domains of world knowledge. Five new mediators were created and added to the CNL-WKR system, spanning from financial information to consumer product information. Finally, the third lesser goal was to create a tool that managed to align a TPTP query with the available external sources. The tool that was created for this was `get_Includes`. This tool was the key to bridging the gap between SPASS-XDB and the external/internal sources; without it, no proofs would be possible.

The overall contribution of this thesis to the science community is a prototype WKR system that already can answer complex queries, but which also provides a vision and direction for future WKR systems to come. The dawn of tedious world knowledge retrieval via search engines is near, and the future lies in WKR systems.

5.3   Future Work

CNL-WKR is by no means a perfect system. As stated before, the main goal was to provide the foundation for a system that works from beginning to end, not to provide a *perfect* system which can understand and answer all user-queries. While the main goal of this research was the former, the larger, general goal was the latter. The future

success of WKR systems relies highly on their ability to compete with search engines and answer queries more quickly and easily than them. Knowledge is something that is universally desired, therefore, accessing knowledge by the quickest means possible is also universally desired.

This section first discusses ideas that did not work, i.e., ideas that seemed promising but in the end were left unused because they did not seem as though they would advance the system. Next, ideas that were out of scope will be discussed. These ideas are topics of research that should be considered in the future in order to improve the system's capabilities of retrieving world knowledge for users.

5.3.1 Failed Ideas

Presented here, are a few ideas that did not work.

When choosing a query language, a number of CNLs were considered. Most were not well documented, while others were not easy enough to use (in consideration for potential users). CELT, on the other hand, was the exception. CELT was considered for some time as a possible query language, but in the end was discarded for ACE because of its lack of consistency while translating controlled English to logic. CELT could translate a large group of sentences, but not with the same amount of success as ACE. It seemed that if CELT had been used, there would have been a lot of issues when submitting queries.

Another idea that didn't work was automating the axiom alignment tool. Initially it was thought that the ideal way to add addition axioms would be through complete

automation. It soon turned out that the alignment would have to be only partially automated. The reason the axiom selection needs to be partially automated is because axioms can only be added if they are relevant to the mediators that are already in place. The tool's full automation is something to be considered for future research.

5.3.2 Out of Scope

Presented here, are a few ideas that were out of this research's scope.

Query Language

One of the larger decisions made in relation to the CNL-WKR system was choosing which query language to use. Although ACE is a great language, and capable of expressing infinitely many sentences, it can still hinder its users. The common problem with any controlled language is that it is controlled and does not allow users the complete freedom to form sentences as naturally as possible. An alternative to using CNLs might be to use an input language similar to search engines. When a user enters a query into a search engine's input field, the query is typically not a complete sentence. Instead, the important words of a query are entered in a loosely-bound structure. For example, here is a query taken from Chapter 4:

Example 5.1 *There is a city and the city is Paris and the city has an elevation.*

In Example 5.1, if a user wanted to find the same information in a search engine, they might type as little as two words, 'Paris' and 'elevation.' The negative side to allowing such a broken language is that the entire meaning of a query can easily

become misinterpreted. Also, it would become impossible to convert such queries into first-order logic, and as a result, reasoning could not occur. The positive side would be that queries would be simpler to process, and because of their simplicity, they would most likely require only a single mediator request.

A solution might be to have two options available, one offering a relaxed setting where users can enter broken, loosely structured text in order to retrieve quick and simple world knowledge facts. The other option offering a strict setting where users can enter complex queries in ACE to retrieve complex world knowledge facts or ontological relations that require multiple reasoning steps or multiple mediators.

Automation of Alignment

Being able to align TPTP translated queries with external sources as well as SUMO, proved to be essential in finding answers. At the moment, the axioms relating to the external sources and SUMO are added by hand, in advance, to `get_Includes`. What would be ultimately desired would be to have this process completely automated. This might prove to be difficult, since an automated tool that can know how to construct first-order logic axioms based on a TPTP query and the available sources, would be hard to design.

What might be feasible though, is automating the axiom selection specifically for SUMO axioms. SUMO contains millions of axioms, which makes automatic selection hard. Despite automatic selection being hard, there has been a considerable amount of research done on proving theories with large axiom sets. This research looks promising as it provides interesting solutions on how to automate the way in which SUMO

axioms could be selected during the inference process of SPASS-XDB.

References

[1] D. B. Lenat, "CYC: A Large-scale Investment in Knowledge Infrastructure," *Commun. ACM*, vol. 38, no. 11, pp. 33–38, 1995.

[2] J. Doyle, "Logic, Rationality, and Rational Psychology - A commentary on Drew McDermott's Critique of Pure Reason," 1987.

[3] A. Pease and N. Siegel, *Sigma Knowledge Engineering Environment Installation Instructions and User Guide for Sigma 2.02.* Articulate Software, October 2007.

[4] G. Antonious and F. van Harmelen, *A Semantic Web Primer.* Cambridge, MA, USA: MIT Press, 2008.

[5] M. Sainsbury, *Logical Forms.* Malden, MA, USA: Blackwell Publishing, 2001.

[6] W. Goldfarb, *Deductive Logic.* Indianapolis, IN, USA: Hackett Publishing Company, Inc., 2003.

[7] E. Mendelson, *Introduction to Mathematical Logic.* Wadsworth and Brooks/Cole, 3 ed., 1987.

[8] G. Sutcliffe, "The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 337–362, 2009.

[9] A. Pease and W. Murray, "An English to Logic Translator for Ontology-based Knowledge Representation Languages," in *In Proceedings of the 2003 IEEE International Conference on Natural Language Processing and Knowledge Engineering*, pp. 777–783, Prentice Hall, 2003.

[10] R. Schwitter, K. Kaljurand, A. Cregan, C. Dolbear, and G. Hart, "A Comparison of three Controlled Natural Languages for OWL 1.1," in *4th OWL Experiences and Directions Workshop (OWLED 2008 DC)*, (Washington), 1–2 April 2008.

[11] K. Kaljurand, "Paraphrasing Controlled English Texts," in *Pre-Proceedings of the Workshop on Controlled Natural Language (CNL 2009)* (N. E. Fuchs, ed.), vol. 448 of *CEUR Workshop Proceedings*, CEUR-WS, April 2009.

[12] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Attempto Controlled English for Knowledge Representation," in *Reasoning Web, Fourth International Summer School 2008* (C. Baroglio, P. A. Bonatti, J. Małuszyński, M. Marchiori, A. Polleres, and S. Schaffert, eds.), no. 5224 in Lecture Notes in Computer Science, pp. 104–124, Springer, 2008.

[13] N. E. Fuchs, U. Schwertel, and R. Schwitter, "Attempto Controlled English — Not Just Another Logic Specification Language," *Lecture Notes in Computer Science*, vol. 1559, pp. 1–20, 1999.

[14] M. R. Genesereth and R. Fikes, "Knowledge Interchange Format: Version 3.0: Reference Manual," Stanford University, California, 1992.

[15] A. Pease, *Standard Upper Ontology Knowledge Interchange Format*. Articulate Software, June 2009.

[16] J. van Eijck, "Discourse Representation Theory," 2005.

[17] E. Kaufmann and A. Bernstein, "How Useful are Natural Language Interfaces to the Semantic Web for Casual End-users?," 2004.

[18] R. F. Simmons, "Natural Language Question-answering Systems: 1969," *Commun. ACM*, vol. 13, no. 1, pp. 15–30, 1970.

[19] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural Language Interfaces to Databases—An Introduction," *Natural Language Engineering*, vol. 1, no. 1, pp. 29–81, 1995.

[20] A.-M. Popescu, O. Etzioni, and H. A. Kautz, "Towards a Theory of Natural Language Interfaces to Databases.," in *IUI*, pp. 149–157, ACM, 2003.

[21] O. A. McBryan, "GENVL and WWWW: Tools for Taming the Web," in *In Proceedings of the First International World Wide Web Conference*, pp. 79–90, 1994.

[22] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks*, vol. 30, no. 1-7, pp. 107–117, 1998.

[23] S. Brin, "Extracting Patterns and Relations from the World Wide Web," in *WebDB*, pp. 172–183, 1998.

[24] de Melo, Gerard and Suchanek, Fabian M. and Pease, Adam, "Integrating Yago into the Suggested Upper Merged Ontology," Research Report MPI-I-2008-5-003, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, August 2008.

[25] F. Suchanek, G. Kasneci, and G. Weikum, "YAGO: A Core of Semantic Knowledge - Unifying WordNet and Wikipedia," in *16th International World Wide Web Conference (WWW 2007)* (C. L. Williamson, M. E. Zurko, and P. J. Patel-Schneider, Peter F. Shenoy, eds.), (Banff, Canada), pp. 697–706, ACM, 2007.

[26] G. Sutcliffe, M. Suda, A. Teyssandier, N. Dellis, and G. de Melo, "Progress Towards Effective Automated Reasoning with World Knowledge," in *Proceedings of the 23rd International FLAIRS Conference* (C. Murray and H. Guesgen, eds.), pp. 110–115, AAAI Press, 2010.

[27] C. Weidenbach, "Combining Superposition, Sorts and Splitting," in *Handbook of Automated Reasoning* (A. Robinson and A. Voronkov, eds.), pp. 1965–2011, Elsevier Science, 2001.

[28] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Discourse Representation Structures for ACE 6.5," Tech. Rep. ifi-2009.04, Department of Informatics, University of Zurich, Zurich, Switzerland, 2009.