Natural Language Parsing of Ambiguous Sentences: Solving Winograd Schemas

Submitted MAY 2014 in partial fulfilment of the conditions of the award of
the degree MSci (Hons) Computer Science
with Artificial Intelligence

George Hallam
goh01u

With Supervision from Brian Logan

School of Computer Science and Information Technology
University of Nottingham

I hereby declare that this dissertation is all my own work,
except as indicated in the text:

Signature ________________

Date _____ / _____ / _____

May 8, 2014

Contents

I Background **1**

1 Introduction **1**
1.1 Brief history of AI . . . 1
1.2 Testing for intelligence . . . 2
1.2.1 Turing test and its problems . . . 2
1.3 Introduction to the Winograd Schema . . . 3

2 Overall aim of the project **4**
2.1 Requirements . . . 4
2.1.1 Evaluation . . . 4

3 Related Work **6**
3.1 Naïve physics . . . 6
3.2 Representations of Common-sense Knowledge . . . 6
3.3 Resolving Complex Cases of Definite Pronouns . . . 7

II System Design & Implementation **8**

4 System design overview **8**

5 Winograd Schema Input **9**

6 Naïve Physics Axioms (The External Knowledge Base) **11**
6.1 Design . . . 11
6.2 Implementation . . . 11
6.2.1 Pour . . . 12
6.2.2 Remove . . . 12
6.2.3 Fit . . . 13
6.2.4 Lift . . . 14
6.2.5 Contain . . . 15

7 Parser (Linguistic Formalisation) **16**
7.1 Research . . . 16
7.1.1 The Stanford Parser . . . 16
7.1.2 Part-of-speech (POS) tagger . . . 17
7.1.3 Simplenlg . . . 17
7.1.4 GrammarScope . . . 18
7.2 Design . . . 18
7.2.1 Creating dependencies . . . 19
7.2.2 Tagging sentences . . . 20
7.2.3 Ensuring words are in the present tense . . . 20
7.3 Implementation . . . 20
7.3.1 Creating dependencies . . . 20
7.3.2 Sentence tagging and tense modification . . . 23
7.3.3 Getting the words from WordNet . . . 23

8 Ontology **24**
8.1 Research . . . 24

8.1.1 WordNet . . . 24
8.1.2 extJWNL . . . 25
8.2 Design . . . 25
8.3 Implementation . . . 26
8.3.1 Retrieving information from WordNet . . . 26

9 Knowledge-base writer **28**
9.1 Design . . . 28
9.1.1 Nouns and adjectives . . . 28
9.1.2 Verbs . . . 29
9.1.3 Creating hypothesis and conjectures . . . 29
9.1.4 Changes to the initial design . . . 30
9.2 Implementation . . . 30
9.2.1 Nouns . . . 30
9.2.2 Verbs . . . 32
9.2.3 Removing duplicates and clearing the memory . . . 32

10 Reasoner **34**
10.1 Design and implementation . . . 34

11 System Evaluation **35**
11.1 Test harness overview . . . 36
11.2 Testing pour . . . 37
11.3 Testing remove . . . 38
11.3.1 Direct knowledge base testing . . . 38
11.4 Testing fit . . . 39
11.5 Testing lift . . . 39
11.6 Testing contain . . . 40

12 Summary and personal evaluation **42**

References **43**

Abstract

The aim of this project is to solve Winograd Schema questions using a common sense knowledge base. Information is extracted from sentences using a parser, then lexical information is gathered from an ontology and appended to a knowledge base containing Naïve Physics axioms. Once this process is complete, the final knowledge base is then reasoned with using a reasoner named Vampire.

Part I
Background

1 Introduction

Common sense is the basic level of practical knowledge and judgment that we all need to help us live in a reasonable and safe way [18]. For any intelligent entity to act sensibly in this world it should be able to apply its own knowledge effectively. Knowledge is possessed by every human even at a young age; generating obvious inferences from this knowledge is known as common sense. From this vast common sense knowledge base, using common sense reasoning enables humans to perform most intelligent activities with ease; speaking, learning, listening, seeing, planning and most important, understanding. Creating a computer program with this common sense knowledge base is the most important goal in the study of artificial intelligence.

Creating common sense is an extremely hard problem. It involves a large base of knowledge with complicated relations and complex reasoning.

> *One morning a little rabbit sat on a bank. He pricked his ears and listened to the trit-trot, trit-trot of a pony. A gig was coming along the road; it was driven by Mr. McGregor, and beside him sat Mrs. McGregor in her best bonnet. As soon as they had passed, little Benjamin Bunny slid down into the road, and set off with a hop, skip, and a jump to call upon his relations, who lived in the wood at the back of Mr. McGregor's garden.*

The paragraph above is the opening lines from 'The Tale of Benjamin Bunny', by Beatrix Potter. The small passage is aimed at young children and every concept within the story is fully understandable to them[1]. The concepts within this short passage involve simple theories of time, space, physics, needs, goals and communication. An intelligent entity needs to know all relevant knowledge from each domain in order to be able to connect the knowledge together.

The power of common sense is often overlooked as it seems obvious to humans. In reality, a complete theory of common sense would contain the fundamental kernel of a complete theory of human knowledge and intelligence [3].

1.1 Brief history of AI

The field of AI was officially founded in summer of 1956 at a conference at Dartmouth Collage. Many attendees including John McCarthy and Allen Newell, became pioneers of AI, creating ground breaking programs that were far beyond what people thought possible. There was great optimism within the field, as shown by the following famous quotes from Herbert Simon "*machines will be capable, within twenty years, of doing any work a man can do*" and Marvin Minsky "*within a generation ... the problem of creating 'artificial intelligence' will substantially be solved*". As a result of underestimating the difficultly of the problem, funding gradually dried up causing research to slow down.

In the 1980's research came alive again with the introduction and success of expert systems, systems that could simulate the knowledge of a human expert. Another boost came again in the 1990's with the use of data mining techniques and unsupervised learning. The increased power of the computer was another huge beneficial factor to the boost. Finally in the recent years systems like Deep blue winning at chess, DARPA's grand challenge leading to Google car, IBM's Watson winning Jeopardy, and Apples Siri are all examples of advances in AI.

It has been over 57 years since Herbert Simon's statement and only now are we starting to scratch the surface of replicating human actions.

[1]Even though it is about a little rabbit named Benjamin

1.2 Testing for intelligence

Alan Turing released the paper *'Computing Machinery and Intelligence'* [30] which posed the famous question *'Can machines think?'* and idea of the *'Turing Test'*. He had a strong belief that holding a conversation with a machine was an important sign of intelligence.

1.2.1 Turing test and its problems

The Turing test uses natural language to test a machines ability to exhibit intelligent behaviour equivalent to that of a human. The premise of the test is simple, a human 'interrogator' is placed in a room alone and is given then job of differentiating between a human and a machine based upon the responses to questions that the interrogator puts forth. After a set amount of tests are completed, the interrogator tries to determine which subject is human and which is a machine. Success is judged on whether or not the machine can be mistaken for a human, tricking the interrogator. However over the years it has been the use of cheap tricks, deceptions and clever responses to passing the test like answering questions like:

How tall are you?
or
Tell me about your parents?

The machine must act out all human behaviours, regardless of if they are intelligent or not, such as: typing mistakes, inability to compute difficult sums, wordplay and emotional outbursts. It also does not test for highly intelligent behaviours, like difficult problem solving. This means if a machine is more intelligent than a human, it must avoid appearing so (fig. 1).

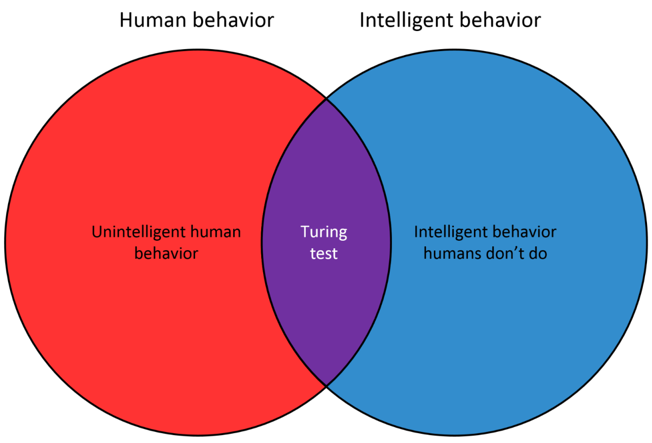

Human behavior
Intelligent behavior
Unintelligent human behavior
Turing test
Intelligent behavior humans don't do

Figure 1: Venn diagram showing the limitations of the Turing test.

It is possible to agree with Turing that the question is whether or not a intelligent behaviour can be attained by a computer program. Using open ended conversation however, may not be the best method to formally test it, as it can allow tricks and wordplay as discussed above. A new test needs to be created to replace or run alongside that of Turing's, one that can measure a basic form of intelligent behaviour: answering ad-hoc questions. Hector J. Levesque's belief is

'...an alternative test based on Winograd schema questions is less subject to abuse, though clearly much less demanding intellectually than engaging in a cooperative conversation (about sonnets, for example, as imagined by Turing).'

1.3 Introduction to the Winograd Schema

The Winograd schema was first suggested in 2011 by Hector J. Levesque, in his paper named 'On our best be behaviour'. His focus in AI is that of question answering such as;

'Should baseball players be allowed to glue small wings onto their caps?'

Levesque believes that answering one shot questions like above is the best way to test intelligence. Keeping questions to a restricted form ensures it does not suffer from the same downfalls as the Turing Test. An explanation is best demonstrated first with an example:

I poured water from the bottle into the cup until it was full
What was full?
The 'bottle' or the 'cup'?

A Winograd schema question is a binary choice question with the following four properties [10]:

1. Two parties are mentioned in the question (both are males, females, objects, or groups).
2. A pronoun is used to refer to one of them ("he," "she," "it," or "they," according to the parties).
3. The question is always the same: what is the referent of the pronoun?
4. Behind the scenes, there are two special words for the schema. There is a slot in the schema that can be filled by either word. The correct answer depends on which special word is chosen.

In the above question the special word used is '*full*' and the other word is '*empty*'. So each WS actually generates two very similar questions.

I poured water from the bottle into the cup until it was ***full****. What was* ***full****?*
The bottle $\times$
The cup ✓

and

I poured water from the bottle into the cup until it was ***empty****. What was* ***empty****?*
The bottle ✓
The cup $\times$

It is this one small difference between the two questions that guards against the cheap tricks that were present in the Turing Test. Levesque writes some basic guidelines to help keep to this restricted form.

Make the questions Google-proof
Access to a large corpus of English text data should not by itself be sufficient.

Avoid questions with common patterns
An example is "Is x older than y?" Perhaps no single Google accessible web page has the answer, but once we map the word "older" to "birth date," the rest comes quickly.

Watch for unintended bias
The word order, grammar and choice of vocabulary all need to be tested for exploitation. Testing for this bias will make sure the questions are not formulated such that they could help a program answer without any comprehension.

2 Overall aim of the project

2.1 Requirements

For this project to be a success it must meet a set of requirements:

- The system must solve the following set of five Winograd Schema's:

  1. I poured water from the bottle into the cup until it was [full/empty]. What was [full/empty]?
  2. I took the water bottle out of the backpack so that it would be [lighter/handy]. What would be [lighter/handy]?
  3. The trophy doesn't fit into the brown suitcase because it's too [small/large]. What is too [small/large]?
  4. The man couldn't lift his son because he was so [weak/heavy]. Who was [weak/heavy]?
  5. The table won't fit through the doorway because it is too [wide/narrow]. What is too [wide/narrow]?

  The system will be trained and designed around the first two schemas, and the other three will be used for testing.

- It must answer these schema questions by reasoning with a common sense knowledge base. The theories contained in this knowledge base must be general enough so sentences in the same domain can also be solved, which means making the axioms as general as possible.

- The answers must not be explicitly placed into the knowledge base.

- Sentences must still be solvable when the key nouns and verbs are replaced with synonyms or like words. For example the following sentences must be solvable:

  – Five men couldn't heave the rocks up the hill because they were so [weak/heavy]. What was to [weak/heavy]?
  – I decanted wine from the bottle into the glass until it was [full/empty]. What was [full/empty]?

  It is important to note, sentences like these are test sentences, and training will only be done on the first two original sentences.

- Information about the sentences must be gathered from an ontology and this information will then be reasoned with using the naïve physics axioms in the knowledge base.

2.1.1 Evaluation

Proving that the concept system works is the single most important aspect. When designing the system two sentences will be used for training and the other three for testing. A testing harness is needed for the finished product, the test can be administered and graded in a simple and fully automated way [10]. So testing will be done in the following way

1. Test the Winograd Schema with no changes to the sentence
2. Change the nouns in the sentence that the special word is related to
3. Change the verb to a synonym of the original verb and change the nouns
4. Rework, and reword the sentence, with a synonym verb and noun

The system will be graded on if it can correctly solve the five schema questions by gathering lexical information from an ontology, then reasoning with it using axioms from a common sense knowledge base. The main claim here is that normally-abled English-speaking adults would be able to pass the test easily.

3 Related Work

In this section I shall discuss the research and related work. The size and complexity required to complete this project meant there will be a need to use existing systems, ones that are reliable, compatible with the requirements and have sufficient documentation or open source repositories. Most of the existing systems to be used shall be discussed in the System Design section to improve clarity.

3.1 Naïve physics

The Second Naïve Physics Manifesto by Patrick Hayes is a rather cynical paper suggesting that people should be more interested in creating large common sense knowledge bases than concerning themselves with the 'Toy problems' of AI. He proposes "*the construction of a formalization of a sizeable portion of common-sense knowledge about the everyday physical world: about objects, space, movement, substances, time etc.*". His paper talks about the 'what is needed' and the 'why it is needed', but not substantially about how to start creating one. He states some important facts and ground rules on characteristics of such a knowledge base. One such knowledge base should have breadth, depth and uniformity [9].

Breath
: It should cover a whole range of everyday physical phenomena, not just novel examples to fit the 'blocks world'. The everyday world is infinitely rich with varying phenomena, so trying to fill the major holes or identify them is hugely important.

Density
: The ratio of facts to concepts needs to be fairly high. Low density formalisations are unable to express enough information about their concepts. This is normally done as it makes it much easier to reason with, but Hayes is not concerned with ease.

Uniformity
: There should be a common formal framework for the whole formalisation, meaning all connections between axioms and frames can be identified and the creation sub-formalisations are not hindered by differing frameworks.

As creating the whole knowledge base is a giant task Hayes introduces the idea of 'clusters'. A cluster can be explained by imagining a graph with tokens[2] as nodes linked by an arc if there is an axiom containing both. The more densely populated areas of the graph can be identified as clusters. By identifying such clusters enables us to obtain isolated sub-theories. Within the project this clustering will be taken advantage of by identifying a suitable cluster and creating theories and tokens within it to solve Winograd Schema's of this cluster/domain.

3.2 Representations of Common-sense Knowledge

Representations of Common-sense Knowledge by Ernest Davis deals with each area of the common-sense domains: quantities, time, space, physics, cognition purposive actions, and interpersonal relations. He deals with the ideas expressed by Patrick Hayes and attempts to formalise many areas. The book contains concepts as well as formal definitions and example systems for many areas of common sense. Some ideas in the book are exceedingly complex and unnecessary for the domain chosen for the Winograd Schema's, such as social interaction and inner workings of the human mind.

[2]A token is a formal symbol, for example we know if we release a stone from a height it will fall with increasing velocity until it hits something, this theory should provide a token that expresses the concept of 'releasing the stone' as well as tokens expressing the ideas of velocity, direction, impact etc...

The chapters on quantities and physics are extremely useful for building a knowledge base with ideas such as;

> *"Water is pouring slowly into a tank at the top and draining out rapidly at the bottom. The height of water in a tank is an increasing function of the volume. Infer that both the volume and the height of the water in the tank is steadily decreasing"*
>
> *"You weigh a letter and determine that it is less than an ounce and requires only a 25 cent stamp. You therefore plan to ax the stamp, but realize that the stamped letter will weigh more. However, reasoning that the weight of a stamp is negligible as compared to the weight of a letter, you conclude that it will not bring the weight of the letter over the limit."*

For both examples Davis gives complex solutions for solving both problems using first order logic and the reasoning behind them. Whilst the solutions themselves are not able to be added directly to the systems knowledge base, the concepts behind them are insightful.

3.3 Resolving Complex Cases of Definite Pronouns

Resolving Complex Cases of Definite Pronouns: The Winograd Schema Challenge is a paper written by Altaf Rahman and Vincent Ng. In the paper they take on the challenge of solving a less strict form Winograd Schema questions using Machine learning and language parsing. They create a large dataset written by undergraduates consisting of 941 sentences based on this less strict form of Winograd Schema and splitting them 70/30 into a training set and testing set respectively.

They employ machine learning to combine the features derived from different knowledge sources, using a ranking-based approach. Given a pronoun and two candidate antecedents, they aim to train a ranking model that ranks the two candidates such that the correct antecedent is assigned a higher rank [20].

For resolving pronouns a selection of techniques are used by deriving linguistic features from a range of components such as.

- Narrative chains; using the example *'Ed punished Tim because he tried to escape.'* Humans resolve *he* to *Tim* by using there own knowledge that someone who is escaping would normally be stopped as what they are doing is wrong. This same type of knowledge can also be obtained using narrative chains. Narrative chains are partially ordered sets of events centered around a common protagonist, aiming to encode the kind of knowledge provided by scripts [24].

- Google; Uses Google hits and statistical analysis on subsections of sentences to rank the likelihood of it being true. An example used is *'Lions eat zebras because they are predators'* would generate queries such as 'lions are predators' and 'zebras are predators'.

The results were impressive with the system guessing the answer correctly 73.05% of the time: this significantly outperforms state-of-the-art resolvers. As significant as this may be, upon closer inspection it shows why this is not actually solving Winograd Schema questions. Firstly Winograd Schema questions are meant to be 'Google proof'. The sentence *'Lions eat zebras because they are predators'* is invalid; searching the terms 'lion' and 'predator' would result in a higher correlation than 'zebra' and 'predator'. Secondly the whole point of the Winograd Schema is to **understand** what is being said – by reasoning with a common-sense knowledge base the system should output a informed decision.

Part II
System Design & Implementation

The following sections contain the research, design and implementation. They have been combined as this project consists of multiple segments – some segments are existing systems, others are written completely from scratch and some are a hybrid of both; due to this nature it is difficult to distinguish where one ends and the other begins. Each self-contained unit has been given its own segment and then ordered in such a way to maximise clarity. The overall general flow of the system is illustrated in Figure 2 and in the text description below.

4 System design overview

The system will take a Winograd Schema which consists of a sentence, a question and two answers. Using a parser, an ontology, a knowledge base and a reasoner, the system will determine which of the two answers are correct. Once the schema question has been inserted into the system, a parser will be used to extract the relevant information from the plain English sentences. This process will be achieved by looking at the key words and relations within a schema sentence and how it relates to the question being asked. The key words extracted from the sentences will be called '*dependencies*', as they are dependent on the root verb of the schema sentence. The parser also serves as a means to make sure all words are fit to be query the ontology. SimpleNLG will be used to ensure words are in the present tense and a POS tagger will be used to classify each dependencies lexical category (Noun, Verb, Adjective, etc...).

The next step is to use an ontology to gather information about all of the dependencies generated by the parser. An ontology provides a shared vocabulary which can be used to model a domain – that is, the type of objects and/or concepts that exist, their properties and relations[1]. One approach would be to use a large ontology such as OpenCyc. Cyc is a extensive ontology and knowledge base containing information about everyday common sense knowledge. OpenCyc is a smaller, cut-down version of Cyc but it is free to use (but not open source) and is available for download [2]. OpenCyc itself is still very large consisting of 239,000 concepts and 2,093,000 facts is that have a a wide breadth but little depth. Due to the fact that OpenCyc is an outdated and smaller version of Cyc means it has a some significant disadvantages: Quite a few objects do not have relevant information about them, meaning there is no way to infer information about them. There is little documentation and you must pay for on-line training (the standard rate being $1,900.00!) meaning it is extremely difficult to start using. It is requires an extremely powerful machine to run. Due to these reasons this approach was decided against and instead an ontology/lexical database named WordNet was chosen. Instead of having one large knowledge base that contains all information, it will be split into two parts. The first part will contain the words and all information relevant to the current Winograd Schema, such as synonym trees and more general terms for a word. The relevant information will be extracted from WordNet using the dependencies gathered by the parser, creating a knowledge base about words, their derived meanings and important features. The second part will be a set of general axioms of common sense physics of a specific domain. Axioms will be used to link meanings of words together, answer the schema questions and will be written in a way to make sure they are broad enough to ensure generalisation yet still be correct. These axioms can be viewed as the 'Laws of Naïve Physics', expressing how objects react with one another and the results of doing so.

The two parts of the knowledge base will be combined along with a hypothesis (a proposal intended to explain certain facts or observations), and a conjecture (an opinion or conclusion formed on the basis of incomplete information). The hypothesis and conjecture will be generated by the parser and based on information learned from the sentence, the question and the answers.

The final combined knowledge base will then be executed by a reasoner which will prove if the hypothesis and conjecture are true based on the axioms present. This information will be passed back into the software and consequently to the user. Due to the nature of its design, once a test sentence has been proven, you will be able to use synonym verbs, different nouns and reworked sentence structures and still be able to derive an answer.

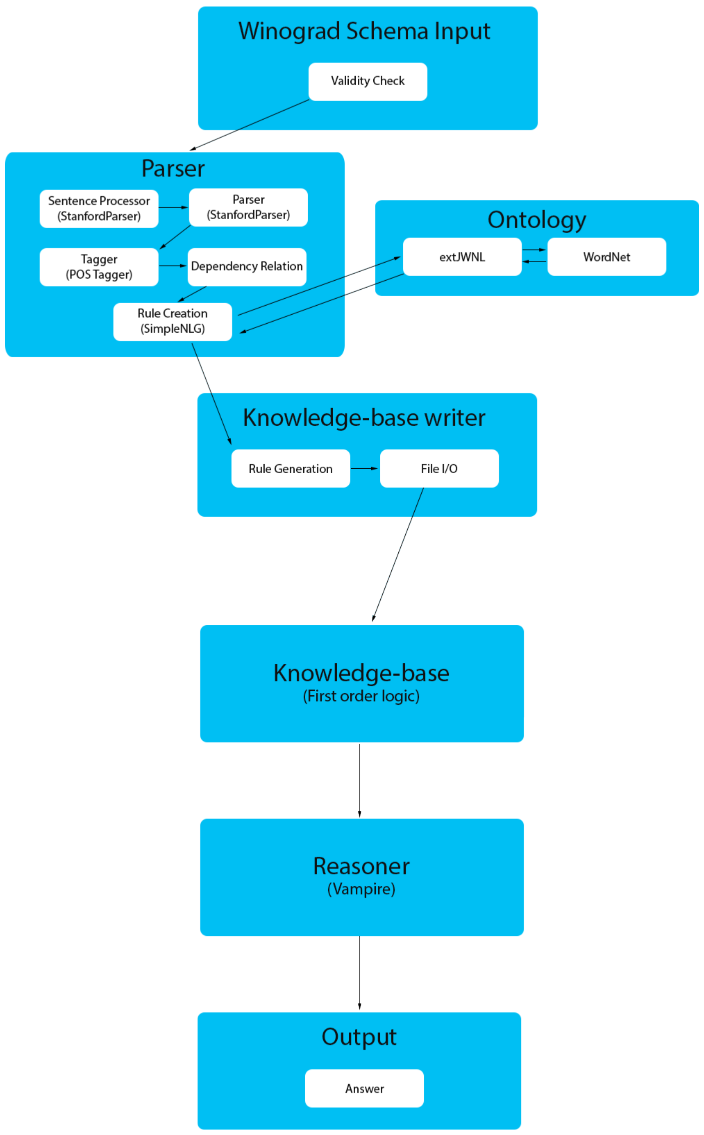


Figure 2: System design layout, with arrows showing how data is passed around

5 Winograd Schema Input

For the system to be usable for the demonstration day some form of user interface is needed. The user interface allows for a high level user to enter a Winograd Schema and should return its results. The input screen for the Winograd Schema is a simple Java JFrame window with the inputs needed to solve the schema question entered.

The window has multiple text boxes for entering: The Winograd Schema sentence, its question, two possible answers and then the choice of the two special words. Below the set of text boxes are two options for increasing the maximum amount of senses that can be written to the knowledge base (see section 9) and switching the conjectures and hypothesis around (see section 11).

To start processing a sentence, none of the text boxes must be left empty and it is up to the user to check spelling. Once the sentence has been processed the result will be displayed in the cyan label at the bottom of the window. If the answer cannot be computed, '*Unknown*' will be displayed, and '*Err*' will be will displayed if any errors are detected. It is important to understand that the objective of this project is not to create a sleek user interface, but it is more research based and still very primitive.

6 Naïve Physics Axioms (The External Knowledge Base)

The external knowledge base is a set of axioms written in first order logic. These axioms enable the system to link information about the words extracted from WordNet to solving the schema questions. The axioms are based on naïve physics and how objects interact with one another. The axioms are written in first order logic then converted to *tptp fof* syntax so Vampire can reason with them. Changing first order logic into *tptp fof* syntax is a trivial task and mostly involves slight syntactic changes. The set of axioms are appended to the end of the knowledge base containing information extracted from WordNet. The naïve physics axioms are kept in a separate file so they can be fully tested and modified without going through every schemas knowledge base. If solving Winograd Schemas of a completely different domain by keeping the naïve physics axioms separate, it enables a choice of what set of 'laws' could be used to solve them.

Creating an knowledge base

6.1 Design

The set of axioms will be designed in such a way that adding a new one will make it possible to infer new information. This is because when new axioms are added they will be related to information currently inside the knowledge base and intertwine. In the Naïve Physics Manifesto by Patrick Hayes says a knowledge base must have three properties: breadth, density and uniformity [9]. The breadth is not currently possible with the time frame, but creating a good base that has both density and uniformity is possible. Having uniformity within the knowledge base ensures for ease of extensibility when adding new axioms in the future. Density within the knowledge base can be ensured by starting with one concept such as, *pouring*, then building upon it, adding related new axioms like a mind map.

Creating a mind map like structure allows new information to be inferred from connected axioms. The knowledge base will start with the initial concept: *pour* as it involves interesting aspects such as the movement of liquids, and their containment. The next step will be adding new related concepts such as *containment*, as pouring a liquid from a container α into another container β will need some theories to govern this idea. After, theories such as *lift* and *remove* will be added to further expand knowledge of naïve physics. As the knowledge base expands, new theories can be added allowing for networks of information. There is no quick way to make the knowledge base, as each axiom has to be written one by one using research from Ernest Davis, Patrick Hayes and dictionary definitions of what the verbs mean.

Each theory will be made as general as possible, because creating a very narrow scoped axioms would result in a much larger knowledge base and cater only for specific problems. It also helps that we are creating a common sense knowledge base, not a specialised one, which allows for even more generalisation. Each theory will first be created as a standalone feature, tested for accuracy and then added to the knowledge base of naïve physics axioms where a testing framework will ensure it does not break existing schema questions. The testing mechanisms will be explained in detail in section 11 on evaluation.

6.2 Implementation

In this section each theory added will be discussed in detail, along with how it works and why it was written that way. Every theory added on top of the initial axioms set *pour* will have its reasons why it was added, in addition to what extra information can be inferred from connecting it to other existing theories.

6.2.1 Pour

Merriam dictionary defines the act of pouring as:

To cause (something) to flow in a steady stream from or into a container or place [14].

From this we can learn a few facts, 'things' that can pour in a steady stream are liquids and fine powders, anything else wouldn't be what people would call a steady stream. Pouring also involves a 'thing' to go from place α to place β, either of which may or may not be a container. If you start pouring something from a container into another container you can instantly assume that one will not be full and the other will not be empty, if this carried on infinitely, eventually this process will stop as one will be empty or the other will be full. Using these English sentences these theories can now be translated into first order logic (and then at a later date to tptp syntax).

Axiom 1:
$pour(x, y, z) \supset liquid(x) \wedge (container(y) \vee object(y)) \wedge object(z) \wedge \neg full(y) \wedge empties(x, y)$.
The axiom $pour(x, y, z)$ means you are pouring a liquid x from one object or container y onto another object z. When you start pouring the liquid out of or off of y it will no longer be full and start to empty liquid x from y.

Axiom 2:
$container(z) \wedge pour(x, y, z) \supset fills(x, y) \wedge \neg empty(z)$.
Similar to that of the previous axiom, however we are now pouring the liquid x into a container z, which would in turn make the container z not empty and the liquid x would fill the container z.

Axiom 3:
$fills(x, y) \wedge infinite\_time \supset full(y)$.

Axiom 4:
$empties(x, y) \wedge infinite\_time \supset empty(y)$.

The use of the predicate *infinite_time* is used as we assume, unless told otherwise, we wait for the action to complete.

6.2.2 Remove

Merriam dictionary defines the act of removing as:

To move or take (something) away from a place [15].

So the act of removing is taking an object away from from somewhere. You can expand on this by saying if you remove something from a container, the container will be lighter, and will not contain the object any more. The word *take*, is used instead of the word *remove*, as it is slightly more general, therefore increasing generalisation:

Axiom 1:
$take(x, y) \supset object(x) \wedge object(y) \wedge lighter(y)$.
The axiom $take(x, y)$ means you are taking an object x from object y. The term lighter is used as currently within the domain chosen we are only concerned about the weight of objects when something is removed. In a future case it would be simple to add extra consequences to this axiom.

Axiom 2:
$container(y) \wedge take(x, y) \supset \neg contains(x, y)$.
If you take an object x from a container y you can say that the container y no longer contains object x.

With these two new axioms we can start to combine *take* with *pour*.

Axiom 2: Modification to axiom 2 of *pour*
$empties(x, y) \wedge infinite\_time \supset empty(y) \wedge take(x, y)$.
With the addition of take, we can infer that when liquid x is emptied out of container y, container y becomes lighter and no longer contains liquid x.

6.2.3 Fit

As we start to remove objects from one thing and place them in another, it is useful to have the verb 'to fit'. The Merriam dictionary defines *fitting* as:

To be the right size and shape for (someone or something)[12]

Fitting can be seen as one thing being able to pass through another. It is important to understand that fitting does not always mean containing; a man can fit through a door way, but that does not mean he is contained in the doorway. To keep *fit* simple we will first say that something can fit inside something else, if it is narrower or the containing object is wider. We can again use the Winograd schema to our advantage as we know one adjective relates to one noun. The axioms used for *fitting* are:

Axiom 1:
$fit(x, y) \supset object(x) \wedge object(y) \wedge canFit(x, y)$.
For an object x to fit into another object y we must see if it 'can fit' – $canFit$ is used because fitting can become a lot more complex the more you look into it. A large inflatable boat can fit into a small box but first it must emptied of air and folded up neatly. This example is not within the domain of this project, however it illustrates the need for multiple theories of '*fitting*'. Therefore, lots of axioms will be needed for fitting as the knowledge base expands. Fit can be used to match words from WordNet and $canFit$ is used as axioms for the theories of fitting.

Axiom 2:
$canFit(x, y) \supset liquid(x) \wedge container(y) \wedge less(volume(x), volume(y))$.
For a liquid x to fit inside a container y its volume of liquid x must be less than container y.

Axiom 3:
$canFit(x, y) \supset wider(y, x)$.
An object x can fit through or in object y if object y is wider than object x. This can be used within the knowledge base as we can take advantage of the binary answers to questions – if something is said to be explicitly wider than something else we assume it will fit.

Axiom 4:
$canFit(x, y) \supset narrower(x, y)$.
An object x can fit through or in object y if object x is narrower than object y. The same idea used above can also be applied here, the system only knows what it is explicitly told.

Axiom 5:

$narrower(x, y) \supset narrow(x) \wedge \neg narrow(y)$.

If object x is narrower than object y, we can assume object x is narrow, and in relation object y is not narrow.

Axiom 6:

$wider(x, y) \supset wide(x) \wedge \neg wide(y)$.

If object x is wider than object y, we can assume object x is wider, and in relation object y is not wide.

As mentioned above, if we are explicitly told that something is narrow, but no mention of how wide something else is, we can assume that the object is narrow enough to fit. General words such as *narrow* and *wide* can be substituted by more specific synonyms, which are able to be generated by WordNet.

6.2.4 Lift

The Merriam dictionary define *lifting* as

To move (something or someone) to a higher position [13].

Lifting is slightly more difficult than the previous terms, as there are limitations to things that can lift or be lifted. Using the dictionary definition to help, we can say object α lifts object β. To simplify aspects slightly we can assume that the object is not screwed down and object α has clear access to object β. There must be a few constraints on both α and β; α must either be a living thing, a vehicle or some form of lifting device and β must be a physical object, as you cannot physically lift an idea etc. For this whole process to take place, the strength of α must be greater than the mass of β – if this is not the case then β cannot be lifted. The following axioms were used for lifting:

Axiom 1:

$$lift(X, Y) \supset ((living\_thing(X) \vee vehicle(X) \vee lifting\_device(X)) \\ \wedge physical\_object(Y) \wedge greater(strength(X), mass(Y))$$

To lift an physical object x, y must be some form of living being, a vehicle or a lifting device. There are other ways to lift objects but these three means are the most general and work well with the chosen domain. Even if y is able to lift its strength must be greater than the mass of object y.

Axiom 2:

$weak(X) \supset less(strength(X), mass(Y))$

If something is weak (in the lifting sense), we can say the strength is less than the mass of a physical object y. $Weak$ is used as it is needed in the domain chosen, and is also very general term.

Axiom 3:

$heavy(X) \supset less(strength(Y), mass(X))$

Axiom 4:

$greater(X, Y) \Leftrightarrow less(Y, X))$

Simple law, if x is greater than y, y must be less than x.

Axiom 5:

$greater(X, Y) \supset \neg greater(Y, X)$

Simple law, if x is greater than y, y can no be greater than x. Useful for negation of terms.

When talking about lifting an object in the terms on the schema questions, when we say something is *weak* we assume it is so weak it cannot lift an object. The same logic is applied to *heavy* if something is heavy and nothing else is said, we assume it is too heavy to lift. Extra axioms can be added here such as:

$$strong(X) \supset greater(strength(X), mass(Y))$$

Due to synonym generation you can then relate synonyms to weak such as puny, feeble and fragile to encompass a larger range without expanding the knowledge base manually.

6.2.5 Contain

The addition of both remove and fit implies the need for rules of containment. The Merriam dictionary defines containment as:

To have (something) inside [11]

The definition is broad, and therefore can be kept general. We can say that contain and fit are synonymous, if an object is being placed inside a container and if the object is smaller. Once an object is inside another object it is said to be ‘contained’ within it.

Axiom 1:
$contain(y, x) \supset fit(x, y)$
y contains x if object x fits in y.

Axiom 2:
$canFit(x, y) \Leftrightarrow smaller(x, y) \wedge container(y)$
We can use the *canFit* axiom to infer containment. Object x can fit into a container y if x is smaller than y.

Axiom 3:
$canFit(x, y) \Leftrightarrow larger(y, x) \wedge container(y)$
Object x can fit into a container y if y is larger than x.

Axiom 4:
$smaller(x, y) \supset small(x) \wedge \neg small(y)$
If x is explicitly said to be smaller than y – We can assume x is small and y in comparison is not small.

Axiom 5:
$larger(x, y) \supset large(x) \wedge \neg large(y)$
If x is explicitly said to be larger than y – We can assume x is large and y in comparison is not large.

Small and *large* are used as they are the most general type of sizing. You can relate small to other words such as; tiny, little, mini, short and so on... The same goes for large. There are other factors that can effect containment but they are not currently needed in this domain.

7 Parser (Linguistic Formalisation)

In linguistic formalisation grammars are explicit descriptions of the rules of a language, usually addressing different levels of granularity. In this section we are not concerned about the sounds of a language, or how those sounds combine to form words. Instead, we are interested in those levels that concern the structure and the meaning of the expressions of a language [19]:

- Syntax studies how words [3] are combined into phrases and sentences. It also defines basic elements and processes that combine these basic elements into more complex structures.
- Semantics investigates the meanings of expressions of a language, and how the meanings of basic expressions are combined into meanings of more complex expressions. Every basic syntactic element is paired with a meaning, and every syntactic rule that combines two elements is paired with corresponding semantic rules that combine their meanings.

7.1 Research

7.1.1 The Stanford Parser

The Stanford Parser is a type of grammar that was first conceptualised in 1969 by Roger C. Schank and Lawrence G. Tesler [25]. The parser described in their paper was purely conceptual. Its primary concern was to *"explicate the underlying meaning and conceptual relationships present in a piece of discourse in any language. With its output being a language free network consisting of unambiguous concepts and their relations to other concepts."*. The parser was based on the Conceptual Dependency model of language [23], the model suggested is a network of linguistic concepts that would fall into one of the following 6 categories:

Governing categories

- PP: An actor or object; corresponding roughly to a syntactic noun or pronoun.
- ACT: In English, corresponds syntactically to verbs, verbal nouns and certain abstract nouns.
- LOC: A noun denoting the location of a conceptualisation.
- T: Denotes the time of a conceptualisation.

Assisting categories

- PA: PP-assister, corresponds roughly to an adjective.
- AA: ACT-assister, an adverb.

The parser works similar to that of how a person would hear a sentence, studying the words one by one. According to the theory;

> *"...as each word is input, its representation is determined and then stored until it can be attached to the concept which directly governs it."*

[3]Although sentences contain words, we are agnostic with respect to what a word is. In the parsing stage we assume a word is a unit that bears meaning and can be mentioned in isolation.

For example, the sentence: "The big man steals the red book" semantic structure can be represented like so:

$$\begin{tikzcd} \mathit{man} \arrow[r, leftrightarrow] & \mathit{steals} & \mathit{book} \arrow[l] \\ \mathit{big} \arrow[u] & & \mathit{red} \arrow[u] \end{tikzcd}$$

Then assigned the following categories:

$$\begin{tikzcd} PP \arrow[r, leftrightarrow] & ACT & PP \arrow[l] \\ PA \arrow[u] & & PA \arrow[u] \end{tikzcd}$$

Today the Stanford parser can automatically extract typed dependencies from English sentences. A typed dependency parser additionally labels dependencies with grammatical relations, such as subject or indirect object [4]. Unlike the conceptual parser which puts words into 6 different categories, the current version has approximately 50 grammatical relations. The dependencies are all binary relations, meaning a relationship exists between a *governor* and a *dependant* [5]. A simple example of the relationship is illustrated in Figure 3.

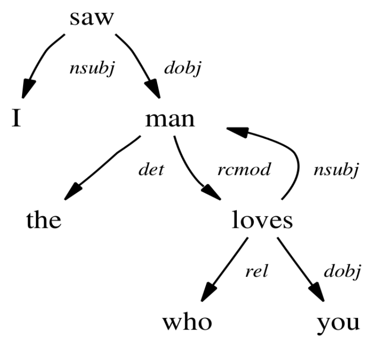


Figure 3: Example of typed dependency parse on the sentence "I saw the man who loves you".

7.1.2 Part-of-speech (POS) tagger

A Part-Of-Speech (POS) tagger is a piece of software that analyses a piece of text and then assigns a token (such as a verb, noun, adjective, etc..) to each word in the sentence. The POS tagger will be a vital component when analysing sentences as it is important for the ontology to know what lexical category a word is. The Stanford's POS tagger is open source and has an accuracy of 97.24% when tagging a sentence [29], when using the Penn Treebank to tokenise words. A 'treebank' is a parsed piece of text that is annotated by its syntactic or semantic sentence structure.

7.1.3 Simplenlg

Simplenlg[4] is a simple Java API developed at the University of Aberdeen. Its primary function is to facilitate the generation of natural language by building and combining phrases. Given a few words, Simplenlg can generate simple sentences. Alongside this, it has a powerful set of syntactic operations for checking and changing the tense of words. The syntactic component covers the full range of English verbal forms, including participals, compound tenses, and progressive

[4]The *'nlg'* stand for Natural Language Generation which is the process of generating natural language from computer based representation such as a knowledge base or first order logic

aspects [6]. By combining Simplenlg and Standford's POS Tagger it is possible to identify and change the tense (if needed) of verbs for the purpose of searching ontologies. This is because research has shown only verbs in the present tense are listed within WordNet.

7.1.4 GrammarScope

GrammerScope is an easy-to-use graphical user interface that extends the Stanford parser enabling inexperienced users to:

- display grammatical relations,
- create type dependency graphs from text,
- modify, import and export grammatical relation definitions.

Its target audience is for individuals teaching English as a second language, as sentence structure and relations are easily shown. The whole of GrammarScope is open source and its libraries are much more concise than that of the standalone version of the Standford parser, making creating typed dependency graphs a matter of calling a single function. It provides a simple-to-use graphic interface to display parsed trees. In addition to, grammatical structure, typed dependencies and semantic graph of any text as parsed by the Stanford Parser/Stanford CoreNLP. Written to Java 1.6 and Swing, it is portable across platforms. GrammarScope was used as a

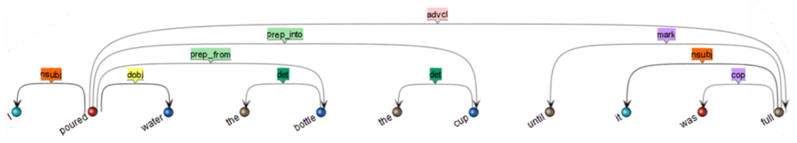


Figure 4: A type dependency graph created in GrammarScope of "*I pored water from the bottle into the cup until it was full*"

proof of concept, to ensure the system to be created was plausible. Creating relational diagrams straight from the Stanford parser involved large amounts of coding and created unclear representations. GrammarScope on the other hand, provides a user friendly UI with tools to explore syntactic relations in sentences with tagging and also creates useful *governor* and *dependant* graphs as shown in figure 4.

7.2 Design

The purpose of the parsing section is to break down the sentences and interpret its underlying meaning, then form connections between words. It then uses this information to create basic rules related around a root word, and builds a list of *dependencies* that can be handed over to WordNet to gather data. A dependency will contain information about two words and how they relate to each other. The parsing section contains multiple intricate pieces that work together in order to build up information about a sentence.
The parser has three main tasks to perform:

1. Creating dependencies by finding the root word of the sentence and what words relate to it.
2. Tag the sentence to find out the lexical category (noun, verb, etc...) that belongs to each word.
3. To make sure all dependencies are in the present tense.

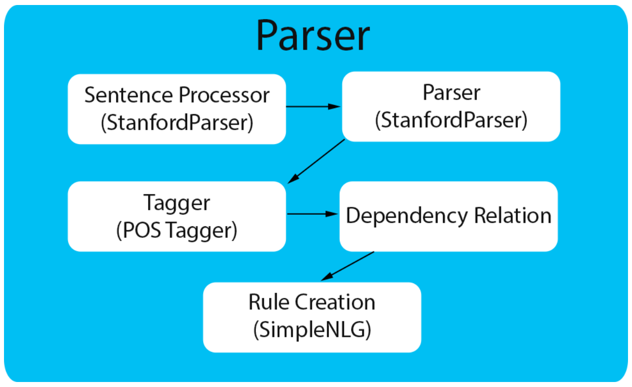


Figure 5: Block diagram of the parsing section of the system

7.2.1 Creating dependencies

The dependency relations are the base for solving the Winograd schemas. Extracting information about how words interact with each other is one of the largest problems in natural language parsing. Due to the nature of the Winograd schemas and how they are a binary questions based on a key words, the sentences all have similar structures. The similar structure of the sentences can therefore be used to our advantage, making the parsing process much less intricate.

Most of the sentences in the chosen domain revolve performing an action (a verb) such as *'pouring'*, *'removing'*, *'lifting'* or *'fitting'* on some object or person causing a result. The Stanford parser uses the most important word as the 'root' when building up dependency trees. From this root node, it works outwards forming different types of relations between them as shown in figure 3. If you assume that the root of the dependency tree is the focus point, you can use the categorised dependent relations provided by The Stanford Parser to locate other key words.

Finding the key word

For this example we can assume the root node is a verb. When dealing with verbs we are looking for 'things' that the verb changes or effects, these 'things' can be summarised by four dependant types:

nsubj : nominal subject

A nominal subject is a noun phrase which is the syntactic subject of a clause. The governor of this relation might not always be a verb: when the verb is a copular verb, the root of the clause is the complement of the copular verb, which can be an adjective or noun.

"Clinton defeated Dole"	nsubj (defeated, Clinton)
"The baby is cute"	nsubj (cute, baby)

prep: prepositional modifier

A prepositional modifier of a verb, adjective, or noun is any prepositional phrase that serves to modify the meaning of the verb, adjective, noun, or even another prepositon. In the collapsed representation, this is used only for prepositions with NP[5] complements.

"I saw a cat in a hat"	prep(cat, in)
"I saw a cat with a telescope"	prep(saw, with)
"He is responsible for meals"	prep(responsible, for)

[5]Noun phrase

dobj : direct object

dobj : direct object The direct object of a VP [6] is the noun phrase which is the (accusative) object of the verb.

"She gave me a raise"	dobj (gave, raise)
"They win the lottery"	dobj (win, lottery)

iobj : indirect object

The indirect object of a VP is the noun phrase which is the (dative) object of the verb.

"She gave me a raise"	iobj (gave, me)

Words that contain one of these dependant types, and are related to the root-node, are words of importance and should be sent to WordNet.

7.2.2 Tagging sentences

To query WordNet one of the items needed its the lexical category of the word; noun, verb or adjective. This can be achieved by running Stanford POS-Tagger on the input sentence. By comparing the the tagged sentence against the list of dependencies it is possible determine the lexical category for each word.

7.2.3 Ensuring words are in the present tense

To query WordNet's database effectively all words queried must be in the present tense. Once a word has been tagged, its tense can be checked by simpleNLG and changed to its present tense form if necessary. The result of this process will be a set of words that describe the sentence, their lexical category and in the present tense. With this information WordNet can now be queried for information.

7.3 Implementation

The implementation stage involves combining existing systems together, then using them to extract the important information from the schema questions. The Stanford Parser's library is extremely large and is capable of some very complex features, making it a powerful piece of software. As in the design section the three main tasks of the parser shall be discussed:

1. Creating dependencies.
2. Tagging the sentence.
3. Ensure all words are in the present tense.

7.3.1 Creating dependencies

The Stanford Parser is already trained with over 60Mb of training data, which must be loaded each time the parser is initialised, making the initial start up a relatively slow process. Once loaded, the first stage in processing is creating a tree of tokens. The raw sentence is tokenised to a list of CoreLabels, then processed into a tree. The resulting tree is shown in figure 6, the actual returned value is a custom tree structure. At this stage it is still difficult to understand the relation between words, but certain syntactic relations are starting to arise such as 'from the bottle' and 'into the cup', which now have their own sub tree.

[6] Verb phrase

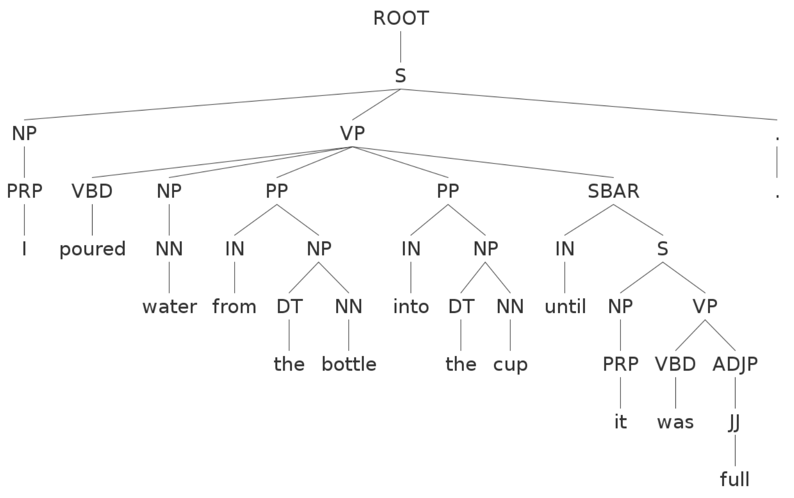
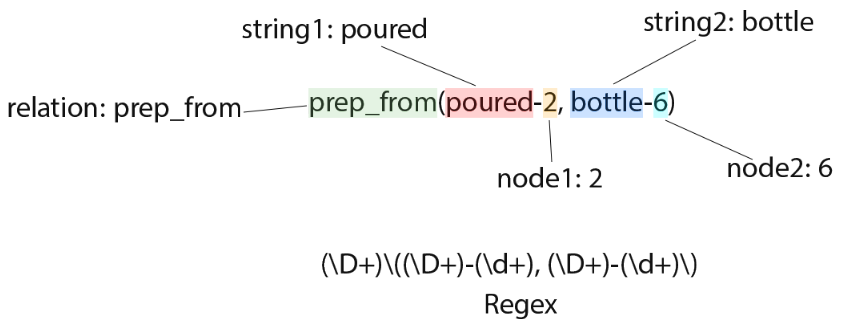


Figure 6: Tokenized tree in a readable format.

The next step is to extract the important information from the tree and use the dependent types to find the relation between words. The Stanford Parser has methods to print trees with different options, but not parse them. By setting the *stdOut* to temporarily write to a string, and by choosing to convert the tree into a ‘collapsed typed dependency’, the correct information can be extracted. The output is shown in figure 7; the green dependencies show useful relations that will be carried forward and used in the next stages of parsing. The relations in red are special cases and will not be used, meaning they will be ignored.

Every typed dependency is then added to an ArrayList in the form of a *Dependency*. The Dependency class takes a string and using a regular expression splits it up into segments:



```
(\D+)\((\D+)-(\d+), (\D+)-(\d+)\)
```

Once all the Dependencies have been segmented and added to the ArrayList, the root node is found and the processes of finding dependencies related to the rule begins. This process has five main steps:

- Check if the relation is allowed: the relation is checked against the predefined list of allowed typed dependency relations, this will make sure only the coloured items shown in figure 7 are allowed.

nsubj(poured-2, I-1)
root(ROOT-0, poured-2)
dobj(poured-2, water-3)
det(bottle-6, the-5)
prep_from(poured-2, bottle-6)
det(cup-9, the-8)
prep_into(poured-2, cup-9)
mark(full-13, until-10)
nsubj(full-13, it-11)
cop(full-13, was-12)
advcl(poured-2, full-13)

Figure 7: Typed dependency list.

- Check if a word is related to the root: any item that is not directly related to the root node is removed, doing this would remove the clause *nsubj(full-13, it-11)*. The ID's of the strings are compared to see if the root node is present in a clause. If the ID is not found the clause is removed.

- Check for negation: negation of a verb is symbolised by the typed dependency '*neg*'. If the negation symbol is found, a flag is set within the Dependency object and will be used at a later time when creating axioms for the knowledge base.

- Check for personal pronouns (optional): personal pronouns are very rarely used in Winograd schemas of this domain, so this option is set by default to false. Setting this to false will remove the clause *nsubj(poured-2, I-1)*, as currently the focus is on what the water is doing, not who poured the water.

- Check for noun compound modifiers (optional): a noun compound modifier is a word that slightly changes a noun such as 'school bus' instead of 'bus' or 'plastic cup' instead of 'cup'. Sometimes this needs to be disabled as the parser may get confused if a sentence is not very well formed.

The outcome of this process is the formation of key words to be used for solving the schema questions and creating the hypothesis axioms to be placed in the knowledge base.

7.3.2 Sentence tagging and tense modification

Sentence tagging is a simple process with extremely high accuracy thanks to Stanford's POS Tagger. The tagger returns a list of lists of tokens in the Penn Treebank format which is then converted to a string for ease of use.

I/PRP poured/VBD water/NN from/IN the/DT bottle/NN into/IN the/DT cup/NN until/IN it/PRP was/VBD full/JJ ./

To extract the token from the word, regular expressions are used. Once the token and word are separated, the definition of the token is analysed, and converted to a base lexical category (noun, verb, adjective). The information is then cross referenced with the final list of dependencies so their lexical category can be set.

SimpleNLG needs two items to check the tense of a word – the word itself and its lexical category. This is why tense checking is the final step in the parsing process. SimpleNLG makes tense manipulation an easy process, no more than a few method calls.

7.3.3 Getting the words from WordNet

The final step in the parsing is combining all the information together to retrieve the words from WordNet. The list of dependencies, with tense modification and lexical categories are used in a method to get words from WordNet. A class named *Word* is created and inside placed the *IndexWord* (extJWNLs means of finding and storing words from WordNet), the words POS tag and how many senses there are of that word. This data can then be used in the following stages for creating trees of information.

8 Ontology

The ontology plays an integral part in the approach to natural language processing. It creates the theoretical basis of the domain with respect to which natural language expressions that will need to be interpreted. While there is no agreed-upon definition of ontologies, there are a number of popular definitions in computer science that are regularly cited by the community:

- an explicit specification of a conceptualisation [7]
- an explicit account or representation of a conceptualisation [31]
- a logical theory accounting for the intended meaning of a formal vocabulary [8]
- a formal explicit description of concepts in a domain of discourse [17]
- a formal, explicit specification of a shared conceptualisation [22]

Ontologies are used in artificial intelligence for modelling knowledge about an entity, their attributes, and their relationships to other entities. Ontologies can be very specific, for example "Disease Ontology", which is a formal ontology of human diseases [26], would not be to useful for solving Winograd Schema questions.

For this project to be successful it relies heavily on a large knowledge base that is both broad and has depth [7]. The following section illustrates how an ontology will be used and integrated into the system.

8.1 Research

8.1.1 WordNet

WordNet is a broad lexical database for the English language [16]. It groups nouns, verbs, adjectives, and adverbs into *synsets*, which are sets of synonyms, each synset describes a unique concept. The lexical database has over 117,000 synsets that are linked to one another. Each synset has a small definition, usually its dictionary definition, so a user knows in which context it is being used. WordNet also records the semantic relations between synsets which results in a large network of meaningful concepts.

As well as containing synonyms of all the words, WordNet also has hypernym and hyponym trees. A hyponym relational tree links a more general synset to a more specific subset (fig 8.). A hypernym relation tree links a more specific synset to a more to its more general superset

$$mammal \rightarrow carnivore \rightarrow canine \rightarrow dog \rightarrow puppy$$

Figure 8: A hyponym tree for mammal, *(... is a kind of mammal).*

(fig 9.). Verb synsets towards the bottom of the synset hierarchy (troponyms) show specific

$$mammal \rightarrow vertebrate \rightarrow chordate \rightarrow animal \rightarrow organism \rightarrow ... \rightarrow entity$$

Figure 9: A hypernym tree for mammal, *(mammal is a kind of ...).*

manners of characterising events like the word "speed". This can be characterised into an order such as: *move-jog-run.* The use of hyper/hyponym trees, synonyms and antonyms enables you to build a smaller, more general knowledge base and use the power of WordNet to inference new relations for more specific words.

[7]As suggested by *The (Second) Naïve Physics Manifesto*

8.1.2 extJWNL

The Extended Java WordNet Library (extJWNL) is a Java API[8] for constructing, reading and amending WordNet dictionaries in WordNet format. It can be used to discover synsets and the morphological information that WordNet contains. The use of extJWNL enables you to communicate with WordNet and its database from within a Java application.

8.2 Design

After careful deliberation and extensive research, WordNet was chosen as an ontology and accessed through extJWNL API. The existence of extJWNL was the main contributing factor for using WordNet, as many other ontologies had stand alone interfaces each with their own language to query the knowledge base. WordNet could now be easily integrated into the system as the Stanford Parser, simpleNLG and POS tagger all had API's to interface with Java. An overview of the design can be seen in figure 10 which is a more detailed segment of the overall design (fig.2)

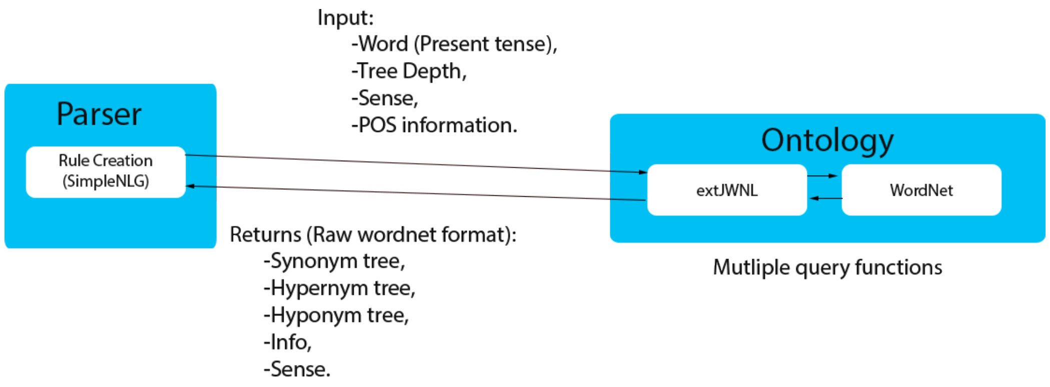


Figure 10: Block diagram of ontology section of the system.

To query WordNet you need two pieces of information with an optional third.

1. A word in the **present tense**. The word itself is the most important, but so is its tense. WordNet has an index of words than can be searched but most of them are only in the present tense form. Querying WordNet with the word *"poured"* for example would return zero results, where as *"pour"* would return all definitions of the word.

2. The word's lexical category; Noun, Verb, or Adjective. Just querying the word alone could lead to the wrong context for example;

   "*The trophy doesn't* ***fit*** *in the brown suitcase because it's to small*"

   In the example above a human would understand *fit* in this context is the verb *to fit*, meaning to 'be of the right shape and size for'. However it could be a noun, meaning 'the particular way in which something, especially a garment or component, fits'. Or an adjective, meaning 'in good health, especially because of regular physical exercise'. Not knowing the words lexical category would result in an unusable system as the wrong lexical trees would be retrieved from the WordNet ontology.

3. The words sense. This is optional due to the design of the program. When a word is found using the correct tense and lexical category, it may have multiple meanings (WordNet calls them 'senses'). Again with an example. Using the Word *'run'* as a verb WordNet returns 41 different senses the top 5 being:

[8] An application programming interface specifies how some software components should interact with each other.

1: (106) run – (move fast by using one's feet, with one foot off the ground at any given time; "Don't run–you'll be out of breath"; "The children ran to the store")

2: (38) scat, run, scarper, turn tail, lam, run away, hightail it, bunk, head for the hills, take to the woods, escape, fly the coop, break away – (flee; take to one's heels; cut and run; "If you see this man, run!"; "The burglars escaped before the police showed up")

3: (21) run, go, pass, lead, extend – (stretch out over a distance, space, time, or scope; run or extend between two points or beyond a certain point; "Service runs all the way to Cranbury"; "His knowledge doesn't go very far"; "My memory extends back to my fourth year of life"; "The facts extend beyond a consideration of her personal assets")

4: (20) operate, run – (direct or control; projects, businesses, etc.; "She is running a relief operation in the Sudan")

5: (10) run, go – (have a particular form; "the story or argument runs as follows"; "as the saying goes...")

Sense is needed in case the sense chosen by default (the top one) is not correct. Fetching more than one sense will enable the program to look through different senses of a word. When the correct one is found it can be reasoned with and the correct information extracted.

Once a word is found information can be passed back to the program along with its respective hyponym, hypernym, and synonym tree, a definition of the word and its sense ID. All of this raw data is transferred back into the parser so it can be sorted into something meaningful.

8.3 Implementation

Data about sentences, questions and answers all need to be retrieved from WordNet so it is vital that the implementation is as flexible as possible. This means the parser has to make sure all of the words given to WordNet are in the correct tense and exist. When retrieving information about a word, four items are required by the system and four pieces of information are retrieved.

8.3.1 Retrieving information from WordNet

The following information is used to query the WordNet database.

Word:

Word is a class that is initially created by the parser. At first it simply contains a lexical category (POS) and a IndexWord; a way to search for words in WordNet through extJWNL.

Hyponym tree depth:

The maximum depth of the hyponym trees, this is extremely useful, and limits the depths of the trees. For most words it is best to set the depth to 1 as knowing one word below it is normally suffice. When searching for nouns such as *water*, having a depth greater than 1 can create interesting results. For example a *lake* is a type of water, so when the depth is set to 1 this would not be a problem and the tree would end there. Going any deeper though WordNet will name every lake in existence, then every sea, pond or body of water it knows about, then in turn add this to the knowledge base leaving Vampire to deal with it.

Hypernym tree depth:

The maximum depth of hypernym tree is one of the most important elements of this project. The depth should always be as large as possible, so that all the information can be known about the word. Making the hypernym tree depth small would miss out vital information such as: a *bottle* is a *container* and is also an *object*. Information such as this is needed by the knowledge base and in turn the reasoner.

Maximum senses:

The maximum amount of senses can be a difficult number to choose as some words can have over 50 different senses. However, a large percentage of the time, the top returned sense is the one that is needed – very rarely is it sense number 2 or 3. Limiting the maximum amount of senses to 6 creates a good balance between knowledge base size and correct context of the word.

The above information is used by a method that is aptly named *getWordInformation* which takes an ArrayList of Words, the tree depths and the maximum number of senses. It returns the same ArrayList of Words but with the following information added:

Synonym, Hypernym, and Hyponym tree ArrayList:

Each of the trees are stored in there own ArrayList, with their index within the list being the sense that they refer to. Storing the trees in this manner enables all the information about one noun, verb or adjective to be placed in one Word object.

Information ArrayList:

The information is a dictionary description of the word, this is mostly used for debug purposes and as a unique identifier in the knowledge base. Similar to the trees, the information about all senses are stored in an ArrayList with the index referring to its sense.

The retrieval of trees from WordNet is made extremely simple through the use of extJWNL. Once the dictionary file has been loaded, to get a tree a simple method call is used (fig. 11). To obtain multiple senses for a word, the sense value is incremented within a loop.

```
public PointerTargetTree getHyponymTree(IndexWord word, int depth, int sense) {
   return PointerUtils.getHyponymTree(word.getSenses().get(sense),depth);
}
```

Figure 11: Getting a hyponym tree from WordNet using extJWNL.

What is returned from WordNet is a custom tree structure which will be parsed at a later date and written to the knowledge base. All information about words are retrieved in the same way, resulting in a system that can be queried from a high level. At the end of this stage in the processing, the result is a list of *Words* containing all the data needed to create a knowledge base to solve the schema questions. The data is currently in an unusable format with other trivial pieces of data attached.

9 Knowledge-base writer

The knowledge base writer is the piece of the system that takes all of the current data and turns it into axioms readable by Vampire. It is also able to search the existing knowledge base and make links between verbs based on hypernym trees. The information produced by the knowledge base writer is gathered solely from WordNet and is the reason the whole system works.

9.1 Design

The knowledge base produced by words from WordNet can be very large, especially if hyponym depth is set to its maximum (see section 8.3.1). Due to its size it is very important to lay out this knowledge base in an ordered manner so it can be easily debugged and can be searched effectively. Splitting it up into segments is therefore vital; each segment will have a time stamp to see when it was added and can be used as a unique identifier. Following this, will be the word, its current sense ID and its dictionary definition – all of this information will be in comment form, and never used by the reasoner. Next will be the trees written in *tptp fof syntax*, each with there own unique identifier. This pattern will continue with each sense of the current word, and throughout the rest of the data.

9.1.1 Nouns and adjectives

Nouns and adjectives will be the simplest to add, as they require little to no modification. When creating the hypernym tree for the word 'water' for example will just be the case of using implication to show:

$$\begin{gathered} water \supset liquid \\ liquid \supset fluid \\ fluid \supset matter \\ \cdots \\ physical\_entity \supset entity \end{gathered}$$

However the knowledge base must be written in *tptp fof syntax*, so in the final format it will look as follows:

```
fof(fluid_matter,axiom,fluid(X)=>matter(X)).
fof(liquid_fluid,axiom,liquid(X)=>fluid(X)).
fof(water_liquid,axiom,water(X)=>liquid(X)).
...
fof(physical_entity_entity,axiom,physical_entity(X)=>entity(X)).
```

Importance of the hypernym tree
For this project to work it is important to know what 'things' are and what they are made of. One of the axioms for '*pouring*' in FOL is as follows:

$$pour(x, y, z) \supset liquid(x) \wedge (container(y) \vee object(y)) \wedge object(z) \wedge \neg full(y) \wedge empties(x, y)$$

To begin with, it is not possible satisfy any of these terms. This is why the hypernym trees are needed. They are trees of knowledge – going up creates broader terms, going down more specialised. So when writing the rules for the knowledge base it is best to keep the rules as broad and as general as possible. Keeping them broad will allow for more schema questions to be answered. The example shown in figure 12 shows an overview of how this is achieved.

The hypothesis being:

$$pour(water, bottle, cup)$$

$$x = water$$
$$water(x) \supset liquid(x) \checkmark$$

$$y = bottle$$
$$bottle(y) \supset vessel(y) \checkmark$$
$$vessel(y) \supset container(y) \checkmark$$

$$z = cup$$
$$cup(z) \supset container(z) \checkmark$$

Figure 12: Shows how hypernym trees are used to match unknown ‘things’ to rules.

9.1.2 Verbs

Verbs are more complex as they are used to replace axioms, and build upon existing ones. This means you can not just add them to the knowledge base as with nouns and adjectives. If verbs were added using the same method as nouns and adjectives it could cause everything to be satisfiable. To be able to use synonyms for verbs the set of axioms in the knowledge base first needs to be analysed, then compared to synonym trees currently held in the system. If a verb in a tree matches an axioms in the knowledge base, they are said to be equal to one another. For example, if we had the sentence:

“Petrol gushed from the tank onto the floor until it was empty.”

We could use the exact same set of axioms used to prove the original *pouring* example (pouring water from the bottle into the cup). Using the list of synonyms given by WordNet for poured and reading in the axioms the following steps can be made:

Step 1. Look for synonym to match to rule in knowledge base

$$gush \supset move \times$$
$$gush \supset flow \times$$
$$gush \supset pour \checkmark$$

Step 2. Look at the arity and terms in the rule

$$pour(x, y, z)$$

Step 3. Use implication to make one imply the other

$$gush(x, y, z) \supset pour(x, y, z)$$

By doing this we can now use *gush* and *pour* synonymously. This means no other information about *gush* will be placed in the knowledge base, all trees will be omitted as everything necessary is contained within the one implication.

9.1.3 Creating hypothesis and conjectures

A hypothesis is something we believe to be true and a conjecture is a conclusion based on incomplete information. So to solve a Winograd schema both a hypothesis and conjecture are required to solve it. The sentence will be the hypothesis; “This is what happened” and the question combined with the special words will be the conjecture; “Now is this true?”.

For example, the dependencies created by the *pour* example in order are *pour*, *water*, *bottle*, and *cup*. By combining these dependencies, ensuring the verb is at the front, we can generate the hypothesis:

$$pour(water,bottle,cup).$$

The next step is creating the conjecture; in this schema the special words are *empty* and *full*. Combining the special words with the subject nouns of the sentence, *bottle* and *cup*, generates four different conjectures:

$$\begin{gathered} empty(bottle) \\ empty(cup) \\ full(bottle) \\ full(cup) \end{gathered}$$

Placing one conjecture and the hypothesis within the knowledge base should show if the statement is true or not. The logistics of the reasoning are discussed in more detail in the reasoning section: 10

9.1.4 Changes to the initial design

Since the initial conception, a few changes had to be made to the knowledge base writer as some problems were encountered. The first was the omission of the hypernym trees for nouns. The reason for this was that they were never used as the rule set was very broad and generalised. In addition, having a set of unnecessary axioms added to the knowledge based slowed down the reasoning process. The second was how different senses were implemented – initially one sense was added, reasoned with, and if it failed the next sense would be added (replacing the previous and so on). This proved slow and cumbersome; it was difficult to understand which specific word was the problem, meaning the correct sense of a word could be removed from the knowledge base entirely. With the removal of the hypernym trees, it allowed for more senses of words to be added without effecting performance. The final was the removal of duplicate information – if four words were added to the knowledge base, each with five senses, the top ends of these hyponym trees were largely the same (more so for nouns):

$$\begin{gathered} object(x) \supset physical\_entity(x) \\ physical\_object(x) \supset physical\_entity(x) \\ whole(x) \supset object(x) \\ unit(x) \supset object(x) \\ whole(x) \supset physical\_object(x) \\ unit(x) \supset physical\_object(x) \end{gathered}$$

This information only needed to be written once, so any duplicate axioms were removed.

9.2 Implementation

The main challenge of implementing the knowledge base from the dependencies was making sure all the strings were in the correct format – one mistake would cause Vampire not to run. Each lexical category had to be compiled differently, with different identifiers and different logical connectives.

9.2.1 Nouns

The hyponym tree is a *List* of *PointerTargetNodeList* as an object can be multiple things. A *PointerTargetNodeList* is a linked list of nodes containing a certain set of information. To get all of the data out of the tree, each list and all the data contained within needs to be fully explored and evaluated. To begin with the link list starts from the top down, and we need information from the bottom up. The steps to obtain information from the list is show in pseudo code in algorithm 1.

```
StringBuilder sb
hyperList ← hypernymTreeArray.get(sense)
for PointerTargetNodeList ptnl : hyperList do
    ptnl.reverse
    nPrevious ← null
    nCurrent ← ptnl.head
    while nCurrent.hasNext do
        nCurrent ← nCurrent.next
        hyperCurArr ← nCurrent.getStrings
        if nPrevious.isNotNull then
            hyperPrevArr ← nPrevious.getStrings
            for String strPrev: hyperPrevArr do
                for String strCur: hyperCurArr do
                    sb.append(“fof(” + strCurr + “_” + strPrev + “,axiom,” + strCurr +
                    “(X)=>” + strPrev + “(X)). \n”)
                end
            end
        end
        else
            for String strCur: hyperCurArr do
                sb.append(“fof(” + str + “,axiom,” + str + “(X)).\n”);
            end
        end
    end
end
```

Algorithm 1: Algorithm showing how the hypernym for nouns are created.

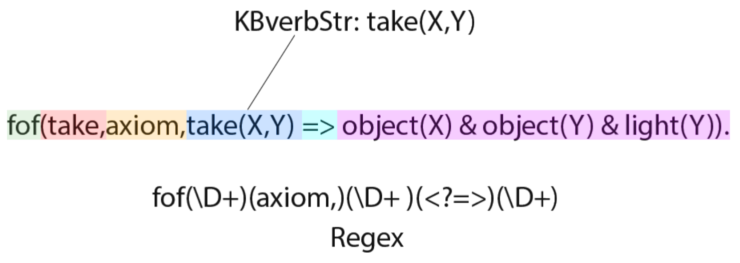
The string appended to the string builder are the axioms that will be added to the knowledge base. The ‘if’ statement that checks to see if the previous node is set to null is used as a base axiom. For example to state *water(water)*, this is needed so that other relations can be formed. A similar method is used for both hyponyms and synonyms, however the string appended to the StringBilder is different.

For hyponyms:
“fof(” + str + “_” + wordString + “,axiom,” + wordString + “(” + wordString + “)” + “=>” + str +“(X)).\n”
Where *wordString* is the word that *str* is a type of:
fof(leaded_gasoline_petrol,axiom,petrol(petrol)=>leaded_gasoline(X)).
fof(leaded_petrol_petrol,axiom,petrol(petrol)=>leaded_petrol(X)).
fof(unleaded_gasoline_petrol,axiom,petrol(petrol)=>unleaded_gasoline(X)).
fof(unleaded_petrol_petrol,axiom,petrol(petrol)=>unleaded_petrol(X)).

For synonyms:
“fof(” + str + “_” + wordString + “,axiom,” + str + “(X)<=>” + wordString + “(X)).\n”
Where *wordString* is synonymous with *str*:
fof(gasolene_petrol,axiom,gasolene(X)<=>petrol(X)).
fof(gas_petrol,axiom,gas(X)<=>petrol(X)).

9.2.2 Verbs

Adding verbs is an in-depth task, it involves comparing every element in all the trees against the axioms in the common sense knowledge base to see if any match. The first stage is to use regular expressions to traverse through the set of axioms in the external knowledge base, as to extract all of the axioms that match the verb.


KBverbStr: take(X,Y)
fof(take,axiom,take(X,Y) => object(X) & object(Y) & light(Y)).
fof(\D+)(axiom,)(\D+)(<?=>)(\D+)
Regex

The axioms from the knowledge base are then checked against the set of synonyms in the verb we are trying to match as shown in figure 13.

Sense 1:
take_away ×
withdraw ×
take ✓

Sense 2:
take out ×
move out ×

Sense 3:
get_rid_of ×

```
rootWordVerbTree $\leftarrow$ rootWord.getVerbTree
rootWord $\leftarrow$ rootWord.getString
for String KBStr: externalKB do
    regex.match(KBStr)
    if regex.isMatch then
        KBStrVerb= regex.group(3)
        verb = KBStrVerb.split("\(+)")
        for String rootStr: rootWordVerbTree do
            if verb.equals(rootStr) then
                newRule $\leftarrow$ (rootStr + verbTerms
                => KBStrVerb)
            end
        end
    end
end
```

Figure 13: An example of how the matching works (left), and pseudo code for finding synonyms (right).

Once the matching process is finished the new set of axioms are added to the knowledge base. There also may be more than one new axiom generated, as words may have more than one meaning.

9.2.3 Removing duplicates and clearing the memory

Removing duplicate axioms is performed using a relatively simple approach, but yields perfect results. When all of the new axioms are generated, they are stored in large StringBuilders until they need to be written to an external knowledge base file so that Vampire can reason with them. When the file writing process begins, it adds each axiom in the StringBuilder line by line, through the use of a *BufferedWriter*. Before writing the axioms to file it checks the string against a *HashMap*. If the strings value in the *HashMap* is null, then the axiom is unique, and is added to the *HashMap* and that line is written to file. If the strings value is not null, then it has already been written, and therefore a duplicate and in turn should not be written to file. Using this method ensures that no two axioms will be the same, but it also ensures no loss of data.

By having a *HashMap* to check for duplicates, it creates a problem when writing multiple files, as the system would think axioms have already been written. In this case, at the end of solving every schema a boolean value can be passed into the method. This ensures the *HashMap* is emptied, and hence solving the problem. Once the information extracted from the words has been written to file, the common sense knowledge base containing the na´ive physics axioms is appended to the end.

10 Reasoner

A reasoner is a piece of software with the ability to infer logical consequences from a set of axioms. The axioms are normally supplied in a knowledge base, as the knowledge base grows in size, so does the time taken to infer logical consequences. Because of this, building my own simple reasoner using forward or backwards chaining would have resulted in an extremely slow and inefficient program.

Vampire
Vampire is an automatic theorem prover for first-order logic [21]. First developed in 1993, it has been rewritten several times and is currently in version 3.0 (released October 2013). It is primarily written by Andrei Voronkov and Krystof Hoder at the University of Manchester. It is well known for being extremely fast, previously winning the World Cup in first-order theorem proving [27]. Other advantages to Vampire include:

- Multi-platform: It can run on MacOS, Windows and Linux allowing for cross platform development, however under Windows it can only run in Cygwin.
- Can deal with large knowledge bases: The amount of information needed for solve Winograd Schema questions would be vast – a reasoner that can handle large KB would be key to solving the questions quickly.
- Supports parallel processing: Takes advantage of multi-core processors running several proof attempts in parallel.

Vampire is simple to use and well documented. To solve a problem all one must do is write the formula in TPTP syntax [28] and run Vampire on the problem. TPTP (Thousands of Problems for Theorem Proves) World is an infrastructure that supports research, development and deployment of Automated Theorem Proving systems. TPTP has a similar syntax to that of Prolog such as variable names starting with upper-case letters. TPTP syntax is understood by all modern first-order theorem provers. The similarity between first-order logic and TPTP can be seen in figure 14.

First-Order Logic	TPTP Syntax
$\top, \bot$	`$false, $true`
$\neg F$	`~F`
$F_1, \wedge ... \wedge F_{\mathrm{n}}$	`F1 & ... & Fn`
$F_1, \vee ... \vee F_{\mathrm{n}}$	`F1 | ... | Fn`
$F_1 \rightarrow F_{\mathrm{n}}$	`F1 => Fn`
$F_1 \leftrightarrow F_{\mathrm{n}}$	`F1 <=> Fn`
$(\forall x_1)...(\forall x_{\mathrm{n}})F$	`![X1,...,Xn]:F`
$(\exists x_1)...(\exists x_{\mathrm{n}})F$	`?[X1,...,Xn]:F`

Figure 14: Correspondence between the first-order logic and TPTP notations

10.1 Design and implementation

Vampire itself is a self-contained piece of software that does not need any modification to work with the software; because of this the design and implementation is extremely simple. Vampire can handle one file at a time and will return refutation if it can prove a set of conjectures and satisfiable if it can not. To deal with a single Winograd Schema, four files will be created each with a different conjecture. Vampire will be invoked through Java's *getRuntime().exec* method with the file location passed through to Vampire. The result will then be read back into Java and handled within the software.

11 System Evaluation

As discussed very briefly in the overview, proving that the concept system works is the single most important aspect. To achieve this a test harness was created in JUnit with bash scripts called by Java to run Vampire on all of the Winograd Schemas. In testing, each schema creates four separate knowledge bases because there are four possible combinations for each schema question. It is best demonstrated with an example:

1. "I poured water from the bottle into the cup until it was full.", "What was full?", "cup"
2. "I poured water from the bottle into the cup until it was full.", "What was full?", "bottle"
3. "I poured water from the bottle into the cup until it was empty.", "What was empty?", "bottle"
4. "I poured water from the bottle into the cup until it was empty.", "What was empty?", "cup"

The above four examples show all combinations of answers for one Winograd schema. In the perfect situation, two sentences will be true and two sentences will be false. In a non-perfect situation, one sentence can be true and three can be wrong, even with this combination the correct answer can still be produced. With having only one of the four sentences being proved however will lower the accuracy.

The system was designed and trained using the following schemas:

- I poured water from the bottle into the cup until it was [full/empty]. What was [full/empty]?
- I took the water bottle out of the backpack so that it would be [lighter/handy]. What would be [lighter/handy]?

Three sentences are left over for testing:

- The trophy doesn't fit into the brown suitcase because it's too [small/large]. What is too [small/large]?
- The man couldn't lift his son because he was so [weak/heavy]. Who was [weak/heavy]?
- The table won't fit through the doorway because it is too [wide/narrow]. What is too [wide/narrow]?

In the evaluation, testing shall be done on all five questions as it is important to see if all the sentences can complete the following four tests:

1. Test the original sentence that was used to create the set of axioms in the knowledge base. There will be no alteration of the sentence from its original form.
2. The two key nouns in the sentence are replaced with two other suitable candidates. These two new nouns must still produce a sentence that makes sense, and have similar if not the same features the nouns they are replacing. In the *pour* example the 'bottle' could be replaced with 'jug'.
3. The two key nouns will be replaced and the verb. The two key nouns will be changed following the same rules as above. The verb will be replaced with a synonym and the sentence structure must not be changed. In this test only three words in the sentence will be changed.

4. The final test involves using two new nouns and a new verb, but the sentence structure can be modified. As long as the verb is kept as a synonym to the original verb, and the whole sentence is related to the initial concept of the schema then this would be classed as valid.

The main claim here is that normally-abled English-speaking adults will pass the test easily. So, if we want to produce behaviour that is indistinguishable from that of people, the software should be also able to pass the test.

11.1 Test harness overview

The implementation of the test harness is simplistic in design and is very scalable as it consists of only two parts – a JUnit test file and a Bash script.

Bash script
The bash script looks in the testing folder where all of the knowledge bases are written, executes Vampire on each one then returns pass or fail depending on Vampires feedback. The script knows the answer is correct by cross referencing the filename which states its predicted output in its title with Vampires return value. If the predicted matches the output then pass is printed to the *stdOut*, if they do not match, 'fail' and the filename of the failed test is printed to the *stdOut*. As Vampire is only available for Linux and runs in Windows through 'Cygwin', it makes sense to use a Bash script as they can be ran from within Java.

JUnit test file
JUnit is a simple framework to write repeatable tests in Java. The test framework was not just used for evaluation, but also for regression testing when changes were made to the code, or to the common sense knowledge base.

The testing script was split into three main parts; an initialisation step, the creation of the knowledge bases, and the final step of running Vampire against the knowledge bases. The initialisation step was achieved using the *@BeforeClass* tag in JUnit, so it would be ran before anything else. The initialisation step consisted of loading all dictionary files into memory and setting up boolean values made specific for testing. Doing this once at the beginning saved large amounts of time, as loading the dictionaries and parsers into memory takes about 5 seconds, and doing this for every single test would take a large time.

The main testing code is multiple test functions that process sentences and give the respective sentences knowledge base specific file names. The snippet of code shown below demonstrates:

```
String s = "Original";
String t = "Pour";
c.setKnowledgeBaseURL("tests/Refutation" + t + "Full"+s+".tptp");
c.processInput("I poured oil from the bucket into the pot until it was full.",
    "What was full?", "pot", "bucket");

c.setKnowledgeBaseURL("tests/Satisfiable"+t+"Full"+s+".tptp");
c.processInput("I decanted wine from the bottle into the glass until it was full.",
    "What was full?", "bottle", "glass");
```

The first two strings represent information about the current test, to be used in the string of the URL the file will be written to. The *setKnowledgeBaseURL* sets the controller to write the next knowledge base file to a specified URL. The file is named in such a way that each URL will be unique and contain information for the bash script to know if the answer is correct or not.

Finally *processInput* takes the sentence, the question, and the two answers and runs the whole software. After each input is processed and the knowledge base is written to file, the internal representation of the knowledge base is cleared so other inputs can be processed. In the final evaluation hundreds of inputs can be processed one after another and all written to unique files.

The final step is accomplished using the *@AfterClass* tag and runs after all testing methods have been completed. This method simply executes the bash script mentioned before and reads it output back into Java to show the results:

```
    p = Runtime.getRuntime().exec("tests/testAllFiles.sh");
    p.waitFor();
```

Due to the nature of Vampire the bash script can only be called on Linux computers.

11.2 Testing pour

The sentence *'I poured water from the bottle into the cup until it was full/empty.'* helped build the whole system, and was also the first set of axioms added to the knowledge base. So testing on this original version of this sentence worked perfectly. There are four sentences that test the *pour* axioms:

Original: "I poured water from the bottle into the cup until it was full/empty."
This sentence contains no modifications.

Noun replacement: "I poured oil from the bucket into the pot until it was full/empty."
The word *water* was replaced with the word *oil*, as they are both liquids. If a non liquid was used the test would have failed. Both *bottle* and *cup* were replaced with two logical containers – a *bucket* and a *pot*.

Verb replacement: "I decanted wine from the bottle into the glass until it was full/empty."
The verb *poured* was replaced with a similar verb *decanted.* To ensure the sentence made sense the *water* was replaced with *wine* and the *cup* with a *glass.* Glass is an interesting replacement, as the first sense of glass is a sheet of glass, such as a window, but the second sense is a glass cup. This shows WordNet can truly create the correct trees for different senses of a word.

Reworked sentence: "Petrol gushed from the tank onto the floor until it was empty."
The whole sentence was restructured but still captures the essence of the original. *Poured* was replaced with *gushed* as it can have multiple meanings such as 'the wind gushed' or in the sense of speaking a large amount 'everyone came up to me and gushed about how lucky I was' and finally the meaning we want 'water gushed from the washing machine'. The noun *tank* replaced *bottle*, as again tank can have multiple meanings which tested WordNets capability of using different senses. The word *cup* was replaced with *floor*, to test a liquid can be poured from a container onto any object. The downside of this is that it is not a full Winograd schema as we can never 'fill the floor', therefore we can only test to see if the tank is empty. To compensate for this when the file is written, it will be labeled as 'satisfiable' as not to produced a false-negative.

The tests were ran on all of the above cases and all passed with 100% accuracy. Using more synonyms for verbs and nouns is not necessary. Running thousands of tests with every container possible and every liquid possible would prove pointless.

11.3 Testing remove

Remove can be seen as an expansion of *pour*, as pouring is only the removal of a liquid. The original sentence for taking/removing is: *'I took the water bottle out of the backpack so that it would be lighter/handy.'*. This sentence was chosen explicitly because of its second meaning. The domain chosen involves reasoning with certain aspects of naïve physics, not about 'things that are useful'. As the knowledge base does not understand what *'handy'* is, it has to rely fully on its understanding of what *'lighter'* and *'taking'* means. The four different sentences used to test the take axioms are:

Original: "I took the water bottle out of the backpack so that it would be lighter/handy." This sentence contains no modifications.

Noun replacement: ''I took the sandwich out of the lunchbox so that it would be lighter/handy." The noun *bottle* was replaced with *sandwich*, which is just a standard object. The *backpack* was replaced with a *lunchbox* which are both containers, however lunchbox is a very common term used in England.

Verb replacement: "I unpacked the clothes out of the suitcase so that it would be lighter/handy." The verb *took* was replaced with *unpacked*, as it is a hyponym of taking. The two nouns *bottle* and *backpack* were replaced with *clothes* and *suitcase* respectively as they were both the same types of entities.

Reworked sentence: "I ladeled soup from the large bowl so it would be lighter/handy." The sentence structure was changed with the base meaning kept the same. The verb *took* was replaced with the obscure word *ladled*, it was purposely used in its past tense as to test simpleNLG's tense modifier. *Ladled* was also used as it is a very specific way of taking something but it is also a noun; 'a long spoon' so this tested the tagger to ensure ladle was tagged as a verb. *Bottle* was replaced by *soup*, for reasons to be discussed in section 11.3.1. Finally *backpack* was replaced with *large bowl*, the adjective *large* was used in front of bowl to test to see if the parser would remove noun compound modifiers and not send them straight through to WordNet.

Even though it was only possible for the knowledge base to understand the concept of 'lighter' and not 'handy', the system was still able to return 100% accuracy. Replacing *took* with any valid synonym retaining sentence structure would ensure accuracy of 100%, only modifying the sentence structure in an obscure way would change this.

11.3.1 Direct knowledge base testing

Testing the knowledge bases full potential through Winograd schemas sometimes proved difficult. However, through direct testing some interesting properties can be shown. Some sentences in the *took* area of testing can be replaced with *pour*, for example changing:

ladle(soup,bowl)
to
pour(soup,bowl)

You are able to ask questions such as:

- light(bowl)
- empty(bowl)
- liquid(soup)

All of the above return refutation and can be proved. As the knowledge base grows, more connections between verbs can be made enabling more complex queries.

11.4 Testing fit

The *fit* examples were the first to test the negation feature and the conjecture switching boolean. Conjecture switching swaps the hypothesis with the conjecture, and the conjecture with the hypothesis – this proves useful when less information is known. The original sentence for fitting is: '*The table won't fit through the doorway because it is too wide/narrow.*'. The knowledge base has a basic understanding of fitting and uses the fact if something is said to be explicitly narrow, it will fit. The four different sentences used to test the fit axioms are:

Original: "The table won't fit through the doorway because it is too wide/narrow."
This sentence contains no modifications.

Noun replacement: "The car won't fit through the gap because it is too wide/narrow."
The noun *table* was replaced with *car*, they are both objects and things you would associate with not fitting somewhere. The second noun, *doorway* was replaced with *gap*, gap was used as it is not always associated with an object, sometimes it is associated more of an 'idea' or a concept of a space or an opening. As *gap* has multiple concepts WordNet was able to produce the correct trees.

Verb replacement: "The car won't go through the gateway because it is too wide/narrow."
The noun *car* was used again, and *doorway* was replaced with *gateway*, a very similar word. The verb *fit* was replaced with *go*, go was used as it can be used in many, many different senses. In WordNet the verb *go* has 30 different senses such as 'go away' and 'how fast does your car go'. The sense we are using it in within this sentence is in the term of fitting, however this is quite an obscure sense. The maximum number of senses for this test has to be expanded to over 30. Changing the maximum number of senses to over 30 creates a relativity large knowledge base, over 400 lines, and can sometimes make connections that are not wanted.

Reworked sentence: "The boxes wouldn't go inside the lorries because they were too wide."
The reworked sentence uses plural nouns and changes the structure of the sentence. *Table* and *doorway* have been replaced with *boxes* and *lorries*. Plurals are used to test if simpleNLG and the Stanford parser can extract the singular and pass it to WordNet successfully.

Initially all four tests failed, as conjecture switching was disabled. Once conjecture switching was enabled, two out of four passed, verb replacement and reworked sentences both failed. They failed as the maximum sense count was limited to 5, once this was set to the higher value of 30, all four tests passed.

11.5 Testing lift

The *lift* example tests WordNets ability to relate specific objects to broader terms, such as a *forklift* to a *vehicle* and a *lifting device*. It also uses conjecture switching and negation like the previous example. So it is testing more and more aspects of the system simultaneously. The original sentence for lift was '*The man couldn't lift his son because he was so weak/heavy.*'. The knowledge base has an understanding of what the requirements to lift something are. The four different sentences used to test the lift axioms are:

Original: "The man couldn't lift his son because he was so weak/heavy."
This sentence contains no modifications

Noun replacement: "John couldn't lift his friend because he was so weak/heavy."
Man was replaced with *John*, as John is a proper noun. Proper nouns can be dealt with

in multiple ways within the software. If a famous person named 'John' is found within WordNet then that is used, however if not, the tree for John would be that of a human. It is also interesting to point out how many obscure words are related to 'John' such as a 'toilet', 'a dead person' and 'an unknown person'. The second noun *son* was replaced with *friend*, as to see if a friend is recognised as an animate thing. 'The' was removed from the beginning of the sentence so it would make grammatical sense.

Verb replacement: "John couldn't move his brother because he was so weak/heavy."
The proper noun *John* was kept from the previous sentence and *son* was replaced with *brother*, a similar concept. The verb *lift* was replaced with *move*, as move has multiple different relations with axioms already present in the knowledge base; move can be related to *pour* and to *remove.* So it is important to test that the correct verb matches the correct axiom.

Reworked sentence: "Five men couldn't heave the rocks up the hill because they were so weak/heavy."
This sentence is by far the most complex of the reworked sentences. It deals with plural objects, negatives, an obscure verb and it is taking place in a location. The addition of the word *hill* tests the parser to make sure words only directly effected by the root are placed into the conjecture. Replacing the noun *man* with *five men*, tests WordNet ability to distinguish between plurals and unnecessary information that does not solve the schema question. *Heave* replaces *lift* as it has multiple meanings, as well as shows how difficult it it to move the rocks. This sentence is unique as it shows how much you can modify the original sentence, not change any axioms but the system still produces the correct results.

All sentences produced the correct answers. However, for noun and verb replacements three out of the four tests produced refutation. When asked 'What was too heavy?' the answer John proved to be refutation complete, which means John was too heavy – this may be true, but it should not stop him from lifting his friend. This problem did not exist in the original or reworked sentences.

11.6 Testing contain

Contains tests both axioms from 'fit' as well as its own axioms about containment. It uses negation and conjecture switching similar to that of the previous example. The original sentence for containment was '*The trophy wouldn't fit in the brown suitcase because it's too small/large.*', the knowledge base needs an understanding of containment and fitting. The four different sentences used to test the containment axioms are:

Original: "The trophy wouldn't fit in the brown suitcase because it's too small/large."
This sentence contains no modifications

Noun replacement: "The football wouldn't fit in the brown bag because it's too small/large."
Trophy was replaced with *football* as they are both objects that one might place somewhere. *Suitcase* was replaced with *bag* as it is a generic container.

Verb replacement: "The phone wouldn't go in the protective case because it's too small-/large."
The *phone* and *case* were used as replacement nouns. In this sentence *brown* was changed to *protective* to see if the parser understands that *protective* modifies the noun *case*, as it is not a normal adjective, and can also be used as a noun. The verb *go* was used as it has many different senses.

Reworked sentence: "The box couldn't contain the ball because it was too small/large."
The nouns *box* and *ball* were used with the new verb *contain*. This checks that *contain* can relate to the *fit* axioms inside the knowledge base, even when the order is changed around.

All sentences produced the correct results as the knowledge base had a clear understanding of containment, fitting and the idea of small and large.

12 Summary and personal evaluation

Overall I feel as though the project was a success. The system was able to solve a set of Winograd schema questions in a chosen domain (Naïve Physics) and answered them with 100% accuracy. One aspect that greatly helped with the success of the project was investing a large amount of time reading papers, books, articles and writing formal proofs and not rushing in. We made this mistake during the group project in second year and it caused lots of unnecessary problems in the later stages of development.

I feel as though this was a challenging project and it has tested both my patience and mental ability. In the first few months particularly there were times at which I felt it was not going to be possible to complete, due to the projects complexity and sheer amount of components. However, with some guidance from my supervisor Brian Logan, I was able to find a substantial amount of papers and books to continue research in the correct direction. This initial guidance was a great help and significantly aided the final outcome of my dissertation.

In the future it would be an interesting task to expand the common sense knowledge base, add more axioms and test different domains. Improvements on the natural language processing could also be made. A book named "*Ontology-Based Interpretation of Natural Language*" was published two weeks prior to the hand in date (hence the reason for omitting it from the research section). The book uses almost the exact same approach as myself, from parsing and reasoning, to the use of an ontology. Having access to this book 8 months ago would have enabled me to produce a much more capable system and would of helped me obtain even better results.

One area I feel as though I was particularly weak in was keeping up with coding practices. As this dissertation was more research based, as I was creating a proof of concept, I felt as if coding standards became less important. This created a large problem when I had a month break over the Christmas holiday to revise. Coming back to uncommented, and honestly badly written code wasted a substantial amount of time and large portions had to be rewritten. Even though the code runs smoothly and is bug free, code was not written as efficiently as it could have been. If the system expands and becomes more complex this may become a more significant issue and have to be dealt with.

If I had the chance to start the project again, I would make a few changes. Firstly I would ensure coding practices where upheld for the entire length of the project and more formal tests were written. Secondly, I would ask a group of people to write a set of Winograd Schema questions in a domain of my choice. Guiding the choice of domain means that I would be able to create a much more thorough and tightly connected knowledge base, where different theories would intertwine much better than the current set. Also, by doing so I would have a much larger set of schema questions, enabling for a larger training set and for a more detailed and comprehensive evaluation.

References

[1] Fredrik Arvidsson and Annika Flycht-Eriksson. Ontologies. `www.ida.liu.se/~janma/SemWeb/Slides/ontologies1.pdf`, 2006.

[2] Cycorp. Opencyc @ONLINE. `http://www.cyc.com/platform/opencyc`, February 2014.

[3] E. Davis. *Representations of Common Sense Knowledge*. The Morgan Kaufmann Series in Representation and Reasoning. Elsevier Science Limited, 1990.

[4] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454, 2006.

[5] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. *URL http://nlp. stanford. edu/software/dependencies manual. pdf*, 2008.

[6] Albert Gatt and Ehud Reiter. Simplenlg: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, pages 90–93. Association for Computational Linguistics, 2009.

[7] Tomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. 1993.

[8] Nicola Guarino. Some ontological principles for designing upper level lexical resources. 1998.

[9] Patrick J Hayes. The second naive physics manifesto. 1985.

[10] Hector J Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, 2011.

[11] Merriam-Webster.com. Contain. `www.merriam-webster.com/dictionary/contain`, 2014.

[12] Merriam-Webster.com. Fit. `www.merriam-webster.com/dictionary/fit`, 2014.

[13] Merriam-Webster.com. Lift. `www.merriam-webster.com/dictionary/lift`, 2014.

[14] Merriam-Webster.com. Pour. `www.merriam-webster.com/dictionary/pour`, 2014.

[15] Merriam-Webster.com. Remove. `www.merriam-webster.com/dictionary/remove`, 2014.

[16] George Miller and Christiane Fellbaum. Wordnet: An electronic lexical database, 1998.

[17] Natalya F. Noy and Deborah L. McGuiness. Ontology development 101: A guide to creating your first ontology. 2001.

[18] Cambridge Dictionaries Online. Common sense. `www.Dictionary.Cambridge.org`, 2011.

[19] Christina Unger Philipp Cimiano and John McCrae. *Ontology-Based Interpretation of Natural Language*. SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES. Morgan and Claypool, 2014.

[20] Altaf Rahman and Vincent Ng. Resolving complex cases of definite pronouns: the winograd schema challenge. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 777–789. Association for Computational Linguistics, 2012.

[21] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI communications*, 15(2):91–110, 2002.

[22] V. Richard Benjamins Rudi Studer and Dieter Fensel. Knowledge engineering: Principles and methods. 1998.

[23] Roger C Schank. *A conceptual dependency representation for a computer-oriented semantics*. Computer Science Department, School of Humanities and Sciences, Stanford University, 1969.

[24] Roger C Schank and Robert P Abelson. *Scripts, plans, goals, and understanding*. Lawrence Erlbaum, 1977.

[25] Roger C Schank and Larry Tesler. A conceptual dependency parser for natural language. In *Proceedings of the 1969 conference on Computational linguistics*, pages 1–3. Association for Computational Linguistics, 1969.

[26] Lynn Marie Schriml, Cesar Arze, Suvarna Nadendla, Yu-Wei Wayne Chang, Mark Mazaitis, Victor Felix, Gang Feng, and Warren Alden Kibbe. Disease ontology: a backbone for disease semantic integration. *Nucleic acids research*, 40(D1):D940–D946, 2012.

[27] Geoff Sutcliffe. The cade-23 automated theorem proving system competition–casc-23. *AI Communications*, 25(1):49–63, 2012.

[28] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The tptp typed first-order form with arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 406–419. Springer, 2012.

[29] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[30] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[31] Michael Uschold. Building ontologies: Towards a unified methodology. 1996.