**CHAPTER 4**

**ANSWER-SET PROGRAMMING**

**4.1 Overview**

The system that has been designed uses an ASP-based approach to represent knowledge from natural language text. So, a basic understanding of answer-set programming is required to understand the remainder of the thesis. This chapter introduces the answer-set programming paradigm and further elaborates on some of the important definitions, concepts and patterns used in answer-set programming. At the end of this chapter, we will go over some of the systems that are developed to run ASP programs.

**4.1 What is Answer-Set Programming (ASP)**

Answer-Set Programming is a declarative problem-solving paradigm that uses both non-monotonic reasoning and logic programming. It is widely used in automatically solving problems relating to representation and reasoning tasks such as modeling reasoning agents, non-monotonic inferences, common sense reasoning, modeling preferences and priorities and many more. An answer set program is a collection of statements that describe the objects of a domain and model relations between them. The semantics of an ASP Program defines a set of possible beliefs that an agent has associated with the program. This set of beliefs is called as an answer-set. The basic constituents of an ASP program are the rules, facts and constraints that describe the problem. Such a program is then passed onto an answer-set solver, which generates answer-sets to the program, that are used to obtain solutions to the problem.

**4.2 Syntax**

In this subsection, we introduce the syntax of an ASP program.

**4.2.1 Atom**

The most basic constituent of the ASP program is an atom. An atomic statement or an atom, is an expression of the form p(t1,…,tn) where p is a predicate symbol of arity n and t1…tn are n terms belonging to the predicate p. Here n >= 0 and the terms ti can be integers or strings of letters, numbers, or underscore that either begin with an underscore or a lower-case letter. If in an atomic statement n = 0, then the brackets are omitted. As an example, ‘parent(mary, alice)’ and ‘alice’ are both atoms, whereas ‘parent(mary, girl(alice))’ is not an atom.

**4.2.2 Literal**

A literal is an atom of the form p(t1,…,tn) or its negation -p(t1,…,tn). Here, -p(t1,…,tn) is referred to as a negative literal. It means that p(t1,…,tn) is false. An atom is called as a ground literal if every term ti in the atom is ground. For example ‘parent(X, Y)’ is a literal where as ‘parent(mary, alice)’ is called as a ground literal.

**4.2.3 Rule or Clause**

An ASP Program consists of a collection of rules of the form

1. l0
2. li <- li+1, …,lm,not lm+1,…,not ln

Here, the symbol ‘not’ is a logical connective and is called as a default negation or negation as failure. Its semantic is discussed later in the chapter. An ASP rule is divided into two parts viz. head and a body. A head is a literal on the left side of the rule and a body is a set of literals on the right side of the rule. The head or the body in a rule can be empty. A rule with an empty head is called as a constraint whereas a rule with an empty body is called as a fact.

**4.3 Semantics**

Using the earlier mentioned syntax, we create an ASP program as a collection of rules, facts, and constraints. In this section, we shall discuss about the meaning of these rules and how they are interpreted while reasoning using these rules. The following are a few semantic patterns commonly used in answer-set programs.

**4.3.1 Modelling Implication**

As we saw earlier, every rule (excluding facts) in ASP has two parts separated by the consequence operator “:-“. In such a rule the head of the rule is said to succeed only if every literal in the body of the rule succeeds. As an example, consider the rule

p :- q, r.

we can read this kind of rule as “if q and r succeed then p succeeds”. Such kind of a pattern is commonly used in ASP programs to show implications.

**4.3.2 Classical Negation**

Classical Negation is a pattern in which we use negative literals, to show the fact that the literal under consideration has been proved to be false. As an example, consider the following rule.

-p(a) :- q(a)

The above rule states that if q(a) is shown to succeed then p(a) is false or -p(a) is true. Classical negation is one of the ways to represent negations in ASP programs.

**4.3.3 Epistemic Disjunction**

We model epistemic disjunctions in ASP when we need to model the semantics for the statement, “Either p(a) succeeds or q(a) succeeds”. Epistemic disjunction is different from exclusive or, where both p(a) and q(a) might succeed at the same time. Thus, to model epistemic disjunction we can make use of even loops in the following manner.

p(a) :- not q(a).

q(a) :- not p(a).

If we solve the above ASP program using an answer-set solver we will get two answer sets {p(a)} and {q(a)}, i.e. either p(a) succeeds or q(a) succeeds.

**4.3.4 Constraints**

Constraints are applied in places where we know for a fact that certain rules are always false and should not be part of the answer-set. As an example, if we know that it is impossible for p(a) to succeed then we can model this constraint as follows

:- p(a).

The above rule states that p(a) is always false. Here we see that a constraint limits the sets of beliefs that an agent has but does not help to derive new information.

**4.3.5 Default Negation or Negation As Failure (NAF)**

Default Negation, also called as Negation As Failure is used to make conclusions based on the absence of information. This type of negation is used to conclude about default rules and assume defaults to be true in case of absence of enough information. As an example, consider the following example where we state that if we are not able to prove that q(a) succeeds then p(a) succeeds.

p(a) :- not q(a).

So in the above rule we assumed that p(a) has succeeded based on the absence of information about q(a). NAF is an important tool to model defaults in ASP programs. Negation as Failure assumes closed-world assumption (CWA), in which we assume, what is not currently know to be true, as false.

**4.4 Default Reasoning**

Default Reasoning or Representing Defaults is one of the advantages of using ASP. The concept of closed-world assumption discussed earlier is an example of default reasoning where we default the value of the literal to fail in the absence of the literal in the answer set. Default reasoning is very useful in modelling human reasoning as we can draw conclusions even in the absence of information by defaulting to the default rule. Default reasoning thus plays an important role in common sense reasoning and understanding. In case of ASP, a default d stated as “Normally elements of class C have property P” is represented as the following rule

p(X) :- c(X),

not ab(d(X)),

not -p(X).

Here, ab(d(X)) can be read as “X is abnormal with respect to the default assumption d” and not -p(X) can be read as “We can’t successfully prove that p(X) is false” or “p(X) may be true”.

Default reasoning uses two kinds of exceptions viz Strong exceptions and weak exceptions. Weak exception makes the default inapplicable and stop the agent from making a default conclusion. For example, in the above-mentioned default rule we can apply a weak exception e(X) by adding the following rule to the program

ab(d(X)) :- not -e(X).

The exception states that X may not be applicable to d if e(X) may be true. Similarly, Strong Exceptions refute the defaults conclusion by allowing the agent to derive the opposite to be true. This can be demonstrated by adding the following rule to the program

-p(X) :- e(X).

The above rule states that p(X) is false if e(X) succeeds, which allows us to defeat d’s conclusion that normally class C elements have the property P.

**4.5 Present systems SASP, CLASP**

TBD

**References:**

1. Knowledge Representation, Reasoning, and the Design of Intelligent Agents (The Answer-Set Programming Approach), *Micheal Gelfond, Yulia Kahl*
2. Dissertation – Goal Directed Answer Set Programming, *Kyle Marple*
3. Answer-Set Programming – A Primer, *Thomas Eiter, Giovambattista Ianni,and Thomas Krennwallner*

**CHAPTER 5**

**SYSTEM ARCHITECTURE**

**5.1 Overview**

This chapter mainly focuses on the various parts of the system and how they interact with each other. It also describes the various sub-components and modules used in generating the knowledge base and goes through the various steps required to answer queries with the help of the generated ASP program.

**5.2 System Architecture**

The system is composed of two main components or sub systems viz. the Knowledge Generation System and the Query Generation System. Both these systems function independent of each other. The architecture comprises of a common resource framework that is shared by both these systems. This chapter will describe all these components in detail in the rest of the chapter.

**5.3 Components of the System**



As illustrated in the figure, the Knowledge Generation System, the Query Generation System, and the Common Resource Framework are the three components of the architecture. The Common Resource Framework consists of Natural Language Processing tools such as Stanford Core NLP Tools, WordNet API as well as modules for preprocessing incoming text. The Knowledge Generation System is mainly responsible for extracting knowledge from a natural language text. For extracting the knowledge from text, this component uses Stanford NLP tools like the POS Tagger, Stanford Dependency Parser, and the Stanford NER Tagger to gain more information about the input text. Apart from these resources it also taps into the vast information that is provided by WordNet and tries to extract information from the same. As currently there are a very few digital resources about verbs in the NLP domain, this component provides a flexible way to add custom information about verbs that would be reusable in many scenarios. Thus, the Knowledge Generation System takes in the natural language passage as input and produces rules in the form of three chunks of information, which can be aggregated together to form an ASP program representing all the extractable knowledge from the source text.

To help answer questions posed in Natural Language, the Query Generation System is used to automatically generate a set of queries that can be used to find solutions from the answer-sets generated by the ASP program. To ask queries to the ASP program we need to provide both the queries as well as the ASP program to an Answer-Set Solver like SaSP or Clasp. The Query Generation System generates multiple queries for a question and arranges them in the order of significance, keeping the more constraint queries before the less constraint ones. Hence, the kind of query that would lead to an answer is also a rough metric as to the quality of the answer. Now let’s dive deep into the various components in the architecture and talk about its sub modules and their interactions.

**5.4 Common Resources Framework**

The Common Resources Framework consists of the following modules as illustrated in the diagram.



**5.4.1 Text Preprocessing Module**

The style of writing in natural language text changes based on the domain, author, title of the text and many other factors. To automate text processing, becomes a very hard task when we must consider all these different writing styles. Thus, for this system we assume certain properties about the incoming natural language text. The text pre-processing module is the first module that the input text passes through and it makes sure that the input text conforms to these assumptions. Some of the assumptions that we make about the incoming text include concatenation of compound nouns and resolution of coreferences. National Aeronautics and Space Administration or NASA is an excellent example of a compound noun. In this example, we assume that the system detects and treats NASA to be a single concept (National\_Aeronautics\_and\_Space\_Administration) as opposed to separate words. Coreference resolution is the task of finding all expressions that refer to the same entity in a text. It plays an important role in higher level NLP tasks and so we assume that the coreferences in the incoming text have already been resolved. Many a times due to informal writing styles, humans miss certain words or assume certain words while reading and writing texts. Working with such informal style of English is hard, so we assume that the incoming text is written in formal English. As this system depends on many NLP tools for semantic resources, it is susceptible to any flaws in these tools. The preprocessing module tries to correct any mis-tagged entries in the text, due to ambiguous wording, with the help of relations given by higher level semantic tools.

**5.4.2 Stanford NLP Core Tools**

Stanford Core NLP Tools is a set of linguistic tools that help in analyzing and understanding natural language text. It consists of a lot of different sub tools that can be pipelined one after the other to analyze a piece of text. It provides solutions to NLP tasks like POS Tagging, Parsing, NER Tagging, Coreference resolution and many more that play a vital role in higher level NLP tasks like text understanding. This system uses the Stanford-Core-NLP version 3.9.1 on the Java Platform and makes extensive use of its POS Tagger, NER Tagger, Stanford Dependency Parsing, and some other tools on the framework to process incoming text.

**5.4.3 WordNet Interface**

One of the important things in text understanding is being able to extract more information about concepts in the passage. This helps the system gain a deeper understanding into a concept. WordNet is one such digital resource that helps in gaining more knowledge about a concept. WordNet is a large lexical database of English. It consists of a large number of concepts grouped into sets of words that are synonyms i.e. synsets. WordNet has thus created a huge network of concepts by linking these synsets based on lexical relations and conceptual-semantics. WordNet's structure makes it a useful tool for computational linguistics and natural language processing. Java WordNet Interface or JWI is a Java library for interfacing with Wordnet created at MIT. With the help of JWI this system interfaces with WordNet and extracts semantic relations like hypernyms, hyponyms, meronyms etc. to gain more information on the passage.

**5.5 Knowledge Generation System**

The Knowledge Generation System deals with the generation of rules from text and extracting information from other sources like WordNet. This system is made up of 3 modules which are described as follows



**5.5.1 Knowledge Extraction from Text**

This module is responsible for generating rules and facts from the passage itself. It uses the various rules and patterns mentioned in Chapter 7 to generate part of the ASP program. The ASP rules generated by this module should contain all the information present in the input passage. The input file that is supplied to this module is assumed to be preprocessed according to the earlier mentioned assumptions.

**5.5.2 WordNet Ontology Generation**

To further understand the concepts mentioned in the input passage, the ontology generation module generates rules regarding disambiguation, hypernyms, and meronyms. Word sense disambiguation forms an integral part of NLP and is also tackled by this module using default reasoning. The Hypernym relation and the Meronym relation along with other relations like Antonyms and Synonyms help capture more semantic information into the ASP program. These topics would be touched upon in detail in the forthcoming chapters.

**5.5.3 Default Knowledge Base**

As mentioned earlier, there is very little digitalized information about the semantics of verbs in the NLP domain. Hence, to gain complete understanding of verbs and their usage, it is required to create rules, describing their complete meaning, manually. The default knowledge base makes it feasible to add knowledge about verbs and nouns by hand. Care must be taken to make sure that the knowledge being added is generic in nature and is reusable for other similar scenarios. With the help of such an increasing knowledge base the system can become more efficient and accurate.

**5.6 Query Generation System**



The Query Generation System is responsible for understanding the question asked in natural language text and converting it into a set of ranked queries, that could be understood by the ASP Solver to answer the question. It is comprised of the following 2 modules.

**5.6.1 Query Understanding**

Questions asked in natural language can be classified into multiple types based on various theories. To classify a question into a specific type requires complete understanding of the question along with the type of answer expected by the question. This module is tasked at finding the various components of the question including the kind of question, based on the ‘Wh’ word and the lexical type and kind of answer expected.

**5.6.2 Query Generation**

Using the information provided by the query understanding module, the query generation module first creates the most constraint query applicable for the question under consideration. This module then starts relaxing certain constraints in the query giving rise to lower quality queries or queries with lower confidence. In the later chapters, we discuss this approach in detail.

**References:**

1. <https://stanfordnlp.github.io/CoreNLP/>
2. Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. [The Stanford CoreNLP Natural Language Processing Toolkit](http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf) In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55-60. [[pdf](http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf)] [[bib](http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.bib)]
3. Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.
4. Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of HLT-NAACL 2003, pp. 252-259.
5. Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005), pp. 363-370. <http://nlp.stanford.edu/~manning/papers/gibbscrf3.pdf>
6. Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser Using Neural Networks. In Proceedings of EMNLP 2014.
7. George A. Miller (1995). WordNet: A Lexical Database for English. Communications of the ACM Vol. 38, No. 11: 39-41.
8. Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.
9. Finlayson, Mark Alan (2014) [Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation.](http://projects.csail.mit.edu/jwi/download.php?f=finlayson.2014.procgwc.7.78.pdf) In H. Orav, C. Fellbaum, & P. Vossen (Eds.), [*Proceedings of the 7th International Global WordNet Conference*](http://gwc2014.ut.ee/proceedings_of_GWC_2014.pdf) (GWC 2014) (pp. 78-85). Tartu, Estonia.

**CHAPTER 6**

**SOFTWARE DEVELOPMENT APPROACH**

**6.1 Overview**

This chapter describes the software development approach taken to build the system. I found out that Test Driven Development is one of the best software development approaches for building a rule-based system. This chapter will elaborate on the various stages involved in test driven development and how its principles help in building a stable rule-based system.

**6.2 Test Driven Development**

Using this approach, a paragraph was divided into multiple tests having a sentence each which was used to develop code for the Knowledge Generation System. Each sentence in the paragraph contributed to my understanding of the grammar rules and patterns in the input text. Test Driven Development, abbreviated as TDD, is a software development process that relies on the repetition of a very short development cycle: first the developer writes an automated test case that defines a desired improvement or new function, then produces a minimum amount of code to pass that test and finally refactors the new code to acceptable standards.

Test Driven Development is known to encourage simple designs that inspires confidence in the code developed under the technique. It is related to the test-first programming concepts of extreme programming. Programmers also apply it to improving and debugging legacy code developed with older techniques. A graphical representation of the typical development cycle can be shown as follows.



**6.3 Steps involved in TDD**

As shown in the above development cycle, there are 5 major stages in a single development cycle of the approach. A cycle in TDD follows the below mentioned steps in sequence.

**6.3.1 Add a test**

In test driven development, each new feature begins with writing a test. This test is supposed to fail as it is written before the feature has been implemented. If the test does not fail, then either the proposed feature is already implemented, or the test is defective. To write such a test the developer must completely understand the specifications and the requirements of the new feature. A developer can accomplish this through use cases and user stories to cover the requirements and exception conditions and can write the test in whatever testing framework is appropriate to the software environment. It could also be a modification of an already existing test case. This is a differentiating feature of test driven development which makes the user focus on the requirements before writing any code.

**6.3.2 Run all tests and see if the new test fails**

This is an important step in TDD, which makes sure that the test harness is working correctly, and the new test does not accidentally pass without requiring writing new code. This step also rules out the possibility that the new test always passes thus making the new test useless. Another important factor in this step is that the test should fail for the right reasons. This makes sure that the test is testing the intended condition and only passes when those conditions are met.

**6.3.3 Write some code**

The next step in this process is to write some code to pass this newly added test. The code written in this step may not be efficient and may pass the test inelegantly. This is acceptable, here as we are going to improve and hone the design in later stages. It is important to note here that the newly added code should only be designed to pass the current test, and no further functionality should be assumed or predicted.

**6.3.4 Run the automated tests all see them succeed**

Once all the automated tests pass with the inclusion of the newly added test and its corresponding code then the developer can be confident that the code meets all the current testing requirements. This is a good point from which we can begin the final step of the development cycle.

**6.3.5 Refactor code**

The entire code can now be refactored to accommodate any new updates and changes required in the design triggered by the addition of new code. By running the entire testing suite, the developer can guarantee that the refactoring has not hampered any functionality in the code. Removing duplicate and dead code is important in software development. This step gives the developer a chance to remove duplicates and improve the code design without affecting code.

**6.3.6 Repeat**

Now, the developer can start with another test case and repeat the cycle to improve the functionality of the system. It is recommended that the size of the edit should always be small, with as few as 1 to 10 edits between each test run. If the new code does not rapidly satisfy a new test or other tests start failing unexpectedly then it is recommended to undo the previous code change as compared to excessive debugging. In such cases, continuous integration helps by providing revertible checkpoints.

**6.4 Development Principles**

There are various principles that help while using test-driven development. These principles include “Keep it simple stupid” (KISS), “You are not going to need it” (YAGNI) and many more. Some of these principles along with their advantages has been discussed below.

By focusing on only writing code for passing a single test, the designs can be cleaner than other approaches. To achieve complex design patterns, tests can be written to generate the design pattern. This helps in keeping the changes short and simple to understand, which allows the developer to focus on what is important. Writing tests first before coding up the functionality has been claimed to have many benefits. It helps the developers think about testing from the outset instead of worrying about it later. Also, writing tests first creates a deeper understanding about the concept or feature in the developers mind which in turn helps in writing better code. Failing the test case first, before implementing its required feature, ensures that the test really works and can catch bugs. Test driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage in the development cycle reinforces the developers mental model of the code and boosts confidence in the code. Keeping the changes to be smaller has proved to have multiple benefits including reduced debugging effort and better understanding of code. These principles if followed correctly enable developers to build large scalable systems without adding to the complexity of debugging and maintaining the system.

**6.5 Testing Framework**