Answering Controlled Natural Language Biomedical Queries using Answer Set Programming

Mohammad Syeed Ibn Faiz
Faculty of Science
University of Western Ontario

October 3, 2011

1 Introduction

Internet hosts a large amount of biomedical knowledge in the form of ontologies. There are means to extract useful information from these ontologies, mostly SQL-like formal query languages. This makes it difficult for people with little or no knowledge about a formal query language. But such people constitute a major portion of the userbase of these ontologies. So a simpler query language is required. Since no language is possibly easier than a natural language for a human being, it is undoubtedly a good choice as a query language. But there are certain difficulties with natural languages, especially when it comes to automated processing. The ambiguities in the vocabulary and grammar of a natural language make it difficult to process and reason about queries in that language. One way to resolve this problem is to modify a natural language in a way so that the natural flavour is still there without any of its ambiguities. Controlled natural languages are such modifications.

A controlled natural language is a subset of a natural language, obtained by restricting the grammar and vocabulary in order to eliminate the ambiguities. One such controlled natural language is Attempto Controlled English(ACE). ACE is a subset of standard English with a restricted syntax and restricted semantics governed by a small set of construction and interpretation rules. Attempto Parsing Engine(APE) parses ACE text unambiguously into a discourse representation structure which is a variant of first-order logic. Therefore it is possible to transform a query given in ACE to a logical formalism that would allow us to do automated reasoning to find answer to that query.

Answer Set Programming (ASP) is one such logical formalism. ASP is a knowledge representation and declarative programming paradigm oriented towards difficult search problems. ASP supports representation of constraints, defaults, aggregates, preferences etc. and allows to automate reasoning in absence of complete information. ASP allows to integrate other technologies, like description logics reasoners and Semantic Web technologies. For example, in [2] the authors illustrates the applicability and effectiveness of using ASP for integrating relevant parts of knowledge extracted from biomedical ontologies in RDF(S)/OWL and answering complex queries related to drug safety and discovery.

In our project we have designed and developed a system for answering queries expressed in ACE by transforming the queries to an ASP program. Similar approach has been undertaken before [1]. They designed and developed a controlled natural language, called BioQueryCNL, for expressing biomedical queries over some ontologies, and introduced an algorithm to convert a query in BioQueryCNL into a program in ASP. Though our project is inspired by their work, there are some positive significant differences.

2 Attempto Controlled English

2.1 Basics

Attempto Controlled English(ACE) is a subset of standard English with a restricted syntax and restricted semantics governed by a small set of construction and interpretation rules. The construction rules determine which sentences belong to ACE. In other words construction rules specify the grammar for ACE. On the other hand interpretation rules eliminate ambiguities. For example, one interpretation rule states that if an adverb can modify the preceding or the following verb then it modifies the preceding one. This rule helps to eliminate ambiguity from a sentence like this - "A customer who enters a card manually types a code", which in ACE has the same meaning as "A customer who manually enters a card types a code". Attempto Parser Engine(APE) converts ACE text unambiguously into a Discourse Representation Structure(DRS), which is a variant of first-order logic.

Discourse Representation Structure derived from an ACE text is returned as

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

The first argument of drs/2 is a list of discourse referents, i.e. quantified variables naming objects of the domain of discourse. The second argument of drs/2 is a list of simple and complex conditions for the discourse referents. The list separator ',' stands for logical conjunction. Simple conditions are logical atoms, while complex conditions are built from other discourse representation structures with the help of the logical connectors negation '-', disjunction 'v', and implication '=>' [3].

A DRS like

```
drs([A,B],[condition(A),condition(B)])
```

is usually pretty-printed as

```
[A,B]
condition(A)
condition(B)
```

The DRS uses a reified, or flat notation for logical atoms. For example, the noun *a man* that would customarily be represented as

```
man(A)
```

is represented in DRS as

```
object(A, man, countable, na, eq, 1)
```

where the predicate card has become a constant argument to a predefined predicate 'object'. As a result ACE only has a fixed number of predefined predicates which allows its developers to conveniently formulate axioms for those predicates.

2.2 Predicates

Discourse Representation Structure has eight predefined predicates. Detailed description of each of them can be found in [3]. Here we briefly introduce some of them so that readers can easily understand the example discussed later in this section.

Object-predicates represent objects introduced by different forms of nouns. They have the following form:

```
object(Ref, Noun, Class, Unit, Op, Count)
```

Property-predicates represent properties that are introduced by adjectives. The references can be either variable or expressions. They can take the following forms:

```
property(Ref1,Adjective,Degree)
property(Ref1,Adjective,Degree,Ref2)
property(Ref1,Adjective,Ref2,Degree,CompTarget,Ref3)
```

Relation-predicates represents relations introduced by *of-constructs* like "symptom of disease". They have the following form:

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

Predicate-predicates represent relations that are introduced by intransitive, transitive and ditransitive verbs. They can have the following forms:

```
predicate(Ref,Verb,SubjRef)
predicate(Ref,Verb,SubjRef,ObjRef)
predicate(Ref,Verb,SubjRef,ObjRef,IndObjRef)
```

Modifier_pp-predicates stand for verb phrase modifiers that are introduced by prepositional phrases.

```
modifier pp(Ref1,Preposition,Ref2)
```

Finally a *query*-predicate points to the object or relation a query was put on.

```
query(Ref,QuestionWord)
```

2.3 Complex Structures

Apart from predicates a DRS can also include complex structures. Among the eight possible complex structures here we introduce only a few.

Classical negation is represented as

```
-drs([A,B],[condition(A),condition(B)]
```

Disjunction is represented as

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

Questions are marked in the DRS by the word question and is internally represented as

```
question(drs([A,B],[condition(A),condition(B)])
```

In nested discourse representation structure, a DRS can occur as an element of the condition list of another DRS. For example,

```
drs([A,B],[condition(A),-drs([],[condition(B)])])
```

2.4 Examples

Now we will look at a few smaple ACE sentences and their corresponding DRSs.

2.4.1 Example 1

ACE Sentence: *A man is mortal.*

DRS:

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

The last two digits seperated by a '/' at the end of each predicate stand for the sentence id and token id respectively. Therefore the *object*-predicate is produced for the token *man*, which is the second token in the only sentence.

2.4.2 Example 2

ACE Sentence: *Which drug cures Asthma?*

DRS:

```
[]
           [A,B]
QUESTION   query(A,which)-1/1
           object(A,drug,countable,na,eq,1)-1/2
           predicate(B,cure,A,named(Asthma))-1/3
```

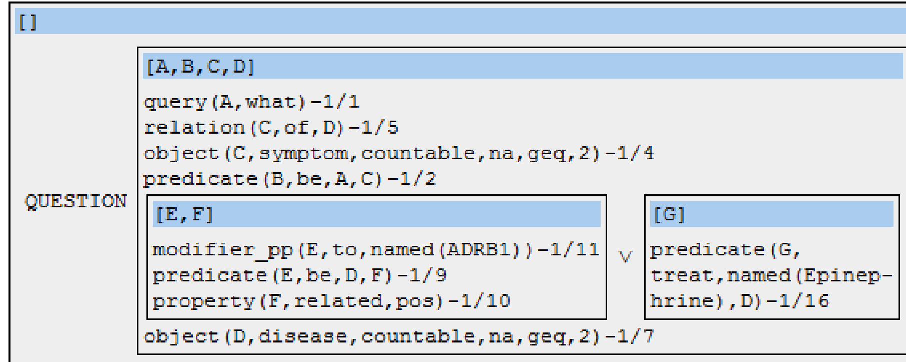
This is an example of nesting of DRS. The textual representation is the following:

```
drs([],[question(drs([A,B],[query(A,which)-1/1,
object(A,drug,countable,na,eq,1)-1/2,
predicate(B,cure,A,named('Asthma'))-1/3]))])
```

2.4.3 Example 3

ACE Sentence: *What are the symptoms of the diseases that are related to ADRB1 or that are treated by Epinephrine?*

DRS:



3 Answer Set Programming

Answer Set Programming (ASP) is a knowledge representation and declarative programming paradigm oriented towards difficult search problems. ASP supports representation of constraints, defaults, aggregates, preferences etc. and allows to automate reasoning in absence of complete information. In ASP knowledge or problem is represented as a program and its meaning is captured by its models, also called answer sets. Answer set for a program can be computed by an answer set solver. There are many answer set solvers including DLV, Smodels, Clasp etc. Though there are differences in syntaxes of these solvers but overall they are similar to some extent. The language of ASP is based on the Prolog language and can be considered as an extended logic programming language.

Traditional logic programming is query driven, that means user enters a query and the system tries to find answer to that query. But ASP works in a different manner. In ASP a problem is encoded into a logic program. The system tries to find models of that program which is the solution to the problem. A model for a logic program is a set of atoms that satisfies all the rules of the program. The following is an example from [4]

```
ide_drive :- hard_drive, not scsi_drive.
scsi_drive :- hard-drive, not ide_drive.
scsi_controller :- scsi_drive.
hard_drive.
```

Here the first rule says that, if we have a *hard_ drive* in our computer and we don't have a *scsi_ drive* then we must have an *ide_ drive*. Then the next rule says that if we have a *hard_ drive* in our computer and we do not have a

ide_ drive then we must have a *scsi_ drive*. The third rule says that if we have a *scsi_ drive* in the computer then we must include a *scsi_ controller* in it. The last rule is a fact which mentions that we have a *hard-drive* in our computer.

This program has two stable models. The first one is:

$$M_1 = \{hard_drive, ide_drive\}$$

and the second one is:

$$M_2 = \{hard_drive, scsi_drive, scsi_controller\}$$

ASP is based on stable model semantics. That means an ASP solver only computes those models that are stable. Informally a model is stable when it satisfies all the rules in the program and when every atom in it has some "reason" to be there: for each atom in the model there must be some rule where that atom is the head of the rule and the rule body is true in the model. This is why in the example shown above the model M_1 does not include *scsi_ controller*. If we add *scsi_ controller* to the first model then the resulting set of model still satisfies all the rules but this model will no longer be a stable model, because *scsi_ controller* is needed only when *scsi_ drive* is there and when *scsi_ drive* is missing there is no reason to add *scsi_ controller* to the model.

Now let's consider an ACE query from the previous section:

> *What are the symptoms of the diseases that are related to ADRB1 or that are treated by Epinephrine?*

This query can be translated into the following rules:

```
what_be_symptom(C) :- symptom_of_disease(C,D),
disease_be_related_to_gene(D,adrb1).
what_be_symptom(C) :- symptom_of_disease(C,D),
drug_treat_disease(epinephrine,D).
```

Now if we include these two rules in an ASP program with all the facts:

```
symptom_of_disease(symptom_n,disease_m)...
disease_be_related_to_gene(disease_x,gene_y)...
drug_treat_disease(drug_p,disease_q)....
```

then the resulting model computed by an answer set solver will provide answer to the given query.

4 Converting ACE Query to Answer Set Program

4.1 Transforming ACE Query into DRS

Throughout this section we will consider the following query:

What are the drugs that treat a disease which causes Anxiety or which is related to HTR1A?

Before transforming a query into DRS we first preprocess the query so that each proper noun is prefixed by its type. For example, after preprocessing the above query will become the following:

What are the drugs that treat a disease which causes Symptom_Anxiety or which is related to Gene_HTR1A?

This type information for each entity is required later on when we construct ASP rules.

Attempto Controlled English text can be unambiguously translated into a Discourse Representation Structure by Attempto Parsing Engine (APE). Using APE 6.6 we obtain the following DRS for the preprocessed query:

```
[]

    QUESTION
    [A,B,C,D,E]
    query(A,what)-1/1
    predicate(E,treat,C,D)-1/6
        [F]
        predicate(F,cause,D,named(Symptom_Anxiety))-1/10
        v
        [G,H]
        modifier_pp(G,to,named(Gene_HTR1A))-1/16
        predicate(G,be,D,H)-1/14
        property(H,related,pos)-1/15
    object(D,disease,countable,na,eq,1)-1/8
    object(C,drug,countable,na,geq,2)-1/4
    predicate(B,be,A,C)-1/2
```

4.2 Parsing Discourse Representing Structure

We have implemented a recursive decent parser that parses a DRS and converts it into an internal representation. This internal representation gives the ability to manipulate every component of a DRS. Our parser parses the following grammar:

```
DRS → drs( Domain , Conditions )
Domain → [] | [ Referent {,Referent}* ]
Conditions → [ Condition {,Condition}* ]
Condition → Predicate | ComplexStructure
Predicate → Object | Property | Relation | Predicate
| Modifier_pp | Modifier_adv | Query
ComplexStructure → Question | Negation | Disjunction
Referent → String
```

```
Object → object( String,String,String,String,String,String )
Property → property( String,String,String )
Property → property( String,String,String,String )
Property → property( String,String,String,String,String,String )
Relation → Relation( String, of, String )
....
Question → question( DRS )
Negation → -( DRS )
Disjunction → v( DRS, DRS )
```

The parser we have implemented is generic, in the sense that its usage is not limited to parsing queries only about biomedical domain.

4.3 Converting Discourse Representation Structure to ASP

Once we obtain an internal representation of the DRS corresponding to the given query we convert it into one or more ASP rules. There can be more than one ASP rule for a given query depending on the query being disjunctive or not. An ASP rule consists of two parts: an atom which is called the head of the rule and a list of atoms called the body of the rule.

4.3.1 Constructing Head Atom

For a Yes/No query the head can be simply a predicate without any variable in its parameter list. For example, let's consider the following query:

Does Epinephrine cure Asthma?

The pretty-printed DRS for this query is the following:

```
[]

    QUESTION
    [A]
    predicate(A,cure,named(Drug_Epinephrine),
     named(Disease_Asthma))-1/3
```

There is no *query*-predicate since this query is not put on any particular entity, rather it merely asks whether a certain relation holds on not. For this query our implementation produces a head like the following:

```
wh(yes)
```

For a Which/What query there is a query predicate in the DRS and the head generation process is more interesting. Let's consider a Which query first:

Which drug cures Asthma?

Corresponding DRS:

```
[]
    QUESTION
    [A,B]
    query(A,which)-1/1
    object(A,drug,countable,na,eq,1)-1/2
    predicate(B,cure,A,named(Disease_Asthma))-1/3
```

For a Which query like this we construct the head atom as follows:

1. Construct an atom H := which(query.ref) //which(A)
2. type := $object_1$.noun where $object_1$.ref = query.ref //type = drug
3. Append type to the name of H
4. Return H //which_drug(A)

Now we consider a What query which is infact a paraphrase of the above query:

What is the drug that cures Asthma?

Corresponding DRS:

```
[]
    QUESTION
    [A,B,C,D]
    query(A,what)-1/1
    predicate(D,cure,C,named(Disease_Asthma))-1/6
    object(C,drug,countable,na,eq,1)-1/4
    predicate(B,be,A,C)-1/2
```

For a What query we construct a head atom using the following algorithm:

1. Find a predicate p such that p.verb="be",p.subjRef = query.ref
2. Create an atom H := what(p.objRef) //what(C)
3. type := $object_1$.noun where $object_1$.ref = p.objRef //type = drug
4. Append type to the name of H
5. Return H //what_be_drug(C)

4.3.2 Constructing Body

Construction of bodies becomes complex in presence of nested DRSs. We consider our internal representation of DRS as a tree. Whenever there is a nested complex structure which is disjunction, the tree has branching. Each branch of that tree contributes a separate body thereby a separate rule. Therefore the number of rules produced for a query is the same as the number of branches in the tree corresponding to its DRS. Our algorithm explores the tree using depth first search and produces body atoms at each leaf of the tree using the following algorithm;

```
1. For each predicate p create an empty atom H
2. <type1,ref1>:= resolveTypeRef(p.subjRef)
3. <type2,ref2>:= resolveTypeRef(p.objRef)
4. adv := getAdverbMod(p.ref)
5. pp := getPrepositionalMod(p.ref)
6. H := type1_p.verb_adv_pp_type2(ref1, ref2)
7. Add H to the list of body atoms
8. For each relation r create an empty atom H
9. <type1,ref1>:= resolveTypeRef(r.ref1)
10. <type2,ref2>:= resolveTypeRef(r.ref2)
11. H := type1_of_type2(ref1, ref2)
12. Add H to the list of body atoms
13. Return list of body atoms
```

Now we will show how this algorithm works for the query we are considering in this section, i.e. the following:

> *What are the drugs that treat a disease which causes Anxiety or which is related to HTR1A?*

The DRS for this query, as shown above, is a tree containing two branches. The left branch corresponds to the following part of the query:

> *What are the drugs that treat a disease which causes Anxiety?*

We now consider this branch. The predicates that we get at the leaf of this branch are the following:

```
query(A,what)-1/1
predicate(F,treat,C,D)-1/6
predicate(E,cause,D,named(Symptom_Anxiety))-1/10
object(C,drug,countable,na,geq,2)-1/4
object(D,disease,countable,na,eq,1)-1/8
predicate(B,be,A,C)-1/2
```

Among these six predicates the first and the last ones have already been considered for generating the head atom. So we will not take them into consideration any more. Among the rest there are two *predicate*-predicates. For each of them we will add an atom to the body.

For the first *predicate*-predicate we get:

```
<type1,ref1>:= <drug, C>
<type2,ref2>:= <disease, D>
H := drug_treat_disease(C, D)
```

For the second *predicate*-predicate we get:

```
<type1,ref1>:= <disease, D>
<type2,ref2>:= <symptom, anxiety>
H := disease_cause_symptom(D, anxiety)
```

Therefore, for the left branch we get the following body:

```
drug_treat_disease(C,D), disease_cause_symptom(D,anxiety)
```

4.3.3 Constructing ASP Rules

When the head atom and all the lists of body atoms are constructed we can generate ASP rules. For each body list we get a separate rule by adding the head atom to the front of the body. For the example query we stated at the beginning of this section, we get the following two rules:

```
what_be_drug(C) :- drug_treat_disease(C,D),
  disease_cause_symptom(D,anxiety)
what_be_drug(C) :- drug_treat_disease(C,D),
  disease_be_related_to_gene(D,htr1a)
```

5 Implementation Issues

We have implemented a system in Java that answers biomedical queries posed in ACE. More specifically it can answer queries involving the following biomedical concepts and the relations between them:

- Gene
- Drug
- Disease
- Symptom

In this section we will briefly describe some issues related to our development.

5.1 Collecting Data from Biomedical Ontologies

To be able to answer biomedical queries our system must have some biomedical knowledge at its disposal. To build this knowledge base we took the help of some biomedical ontologies available on the internet. The knowledge of relationships between genes, diseases and drugs was obtained from PharmGKB [5], which is a Pharmacogenetics and Pharmacogenomics Knowledge Base. That gave us more than 23,000 relationship information in the following format:

...
Gene:PA34913 S100P Drug:PA450218 levonorgestrel PMID:16157482
Gene:PA27504 TSC22D3 Drug:PA450218 levonorgestrel PMID:16157482
Gene:PA29415 HP Drug:PA451900 vitamin e PMID:20415560

To make this knowledge accessible from an ASP program we encoded them into ASP facts:

```
...
gene_drug(s100p,levonorgestrel).
gene_drug(tsc22d3,levonorgestrel).
gene_drug(hp,vitamin_e).
```

We obtained disease and symptom information from Medical Symptoms and Signs of Disease web page [6]. From their web pages we were able to extract about 7,000 relationships between diseases and symptoms.

5.2 Post-Processing

We need a post-processing step to post-process the generated ASP rules. We need this step to make sure that the names of the generated predicates match with the names we have used to encode the biomedical knowledge. We also need to make sure that the parameters of the predicates are in correct order. For example, the relationship between drug and gene is encoded in the knowledge base using atoms with name - "gene_drug". Therefore when we find an atom in a generated ASP rule as the following

```
drug_be_related_to_gene(D, G)
```

we need to rename it as well as to change the order of the parameters as follows

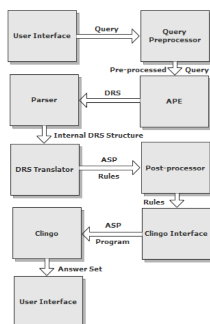
```
gene_drug(G, D)
```

5.3 Answer Set Solver

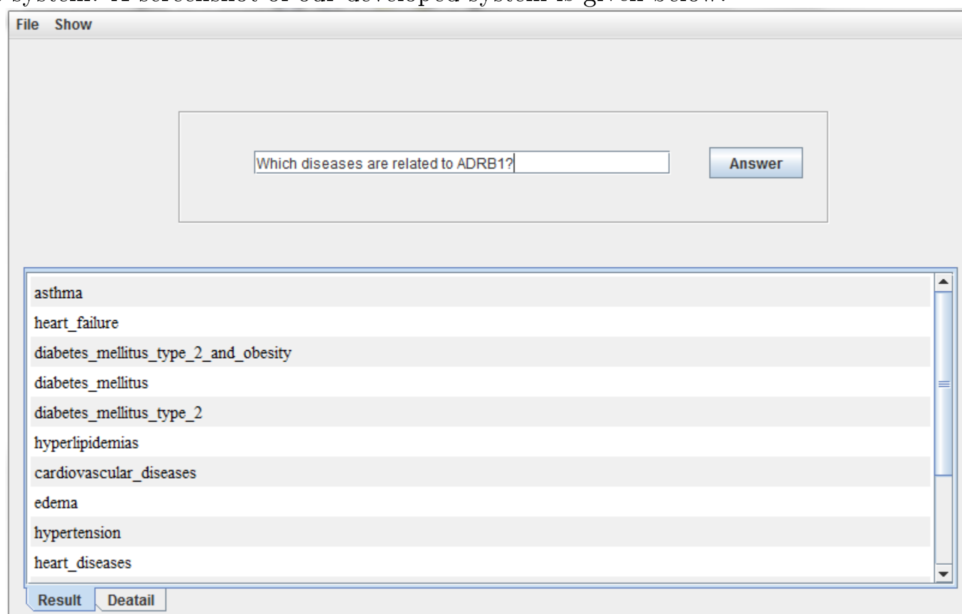
We have used Clasp [7], which is an answer set solver for (extended) normal logic programs. Current answer set solvers work on variable free programs. Therefore, a grounder is needed that, given an input program with first-order variables, transforms it into an equivalent ground (variable-free) program. Gringo is such a grounder. In our implementation we have used Clingo, which combines Clasp and Gringo in a monolithic way. Its input language is that of Gringo, and its output corresponds to that of Clasp.

5.4 System Architecture

The following figure shows the overall system architecture of our developed system:



We have developed a graphical user interface for the convenience of using the system. A screenshot of our developed system is given below:



6 Related Works

Answer Set Programming is used to answer complex queries over biomedical ontologies in [2]. They introduced a new method for integrating relevant parts of knowledge extracted from biomedical ontologies and answering complex queries related to drug safety and discovery. An extension of their work is done in [1]. Here they introduced a controlled natural language for biomedical queries, called BioQueryCNL and presented an algorithm to convert a biomedical query in this language into an answer set program. However, they have not made an implementation of their proposed system publicly available.

Our work in this project is greatly influenced by this latter work. However our implementation is different from theirs in some respect. We have not

designed a query language over ACE like BioQueryCNL. Our implementation supports more forms of query. For example, a query which is not started with a question word is not a valid query in BioQueryCNL. But such a query is valid in ACE and therefore also valid in our implementation. One such example is the following:

Asthma is cured by which drug?

Moreover, our implementation supports query with negation and it can handle arbitrarily large queries where there can be nested negation inside another negation and so on. Support for queries with negation is absent in BioQueryCNL.

One feature that is lacking in our implementation but which is present in BioQueryCNL is the support for cardinality constraints like "at least", "at most", "exactly" and so on.

7 Conclusion

We have designed and developed a system that can answer biomedical queries presented in Attempto Controlled English by transforming them into answer set programs and then solving those programs using an answer set solver. Some example queries that can be answered by our implementation are listed in Appendix A. However, there are certain limitations in our implementation. The knowledgebase of our implementation is not exhaustive and includes only simple relationships between four biomedical concepts, namely gene, drug, disease and symptom. There are ontologies that store a multitude of information. It would be more effective if we can integrate these ontologies in their existing format without the need to encode them.

Another limiation of our implementation is that it does not support queries with cardinality constraints like "at least", "at most", "exactly" and so on. ASP supports aggregates, constraints and preferences. Therefore queries with the cardinality constraints can be conveniently expressed using answer set programs. Future works can be directed towards overcoming these limitations.

Source code and executables of our implementation are available at `http://www.csd.uwo.ca/People/gradstudents/mibnfaiz/bqas/`.

References

[1] Esra Erdem, Reyyan Yeniterzi. Transforming Controlled Natural Language Biomedical Queries into Answer Set Programs. In Proceedings of the Workshop on BioNLP.

[2] Oliver Bodenreider, Zeynep Hande Coban, Mahir Can Doganay, Esra Erdem, and Hilal Kosucu. A preliminary report on answering complex queries related to drug discovery using answer set programming. In Proceedings of ALPSWS.

[3] Norbert E. Fuchs, Kaarel kaljurand, and Tobias Kuhn. Discourse representation structure for ace 6.6. Technical report IFI-2010.0010, Department of Informatics, University of Zurich.

[4] Tommi Syrjanen. Lparse 1.0 User's Manual.

[5] PharmGKB. http://www.pharmgkb.org

[6] MedicalSymptomsSignsDisease. http://www.medicinenet.com/symptoms_and_signs/article.htm

[7] Clasp. http://www.cs.uni-potsdam.de/clasp/

[8] Attempto. http://attempto.ifi.uzh.ch/site

Appendix

A Some Queries and the corresponding generated Answer Set Programs

ACE Query	**Answer Set Program**
Which drug cures Asthma? Asthma is cured by which drug? What is the drug that cures Asthma?	which_drug(A) :- drug_cure_disease(A,asthma).
Does Formoterol cure Asthma? Is Asthma cured by Formoterol?	wh(yes) :- drug_cure_disease(formoterol,asthma).
Which symptoms are alleviated by Epinephrine? Epinephrine alleviates which symptoms? What are the symptoms that are alleviated by Epinephrine?	which_symptom(A) :- drug_alleviate_symptom(epinephrine,A).
What are the symptoms of the diseases that are related to ADRB1 or that are treated by Epinephrine?	what_be_symptom(C) :- symptom_of_disease(C,D), disease_be_related_to_gene(D,adrb1). what_be_symptom(C) :- symptom_of_disease(C,D), drug_treat_disease(epinephrine,D).
What are the drugs that treat a disease which causes Anxiety or which is related to HTR1A?	what_be_drug(C) :- drug_treat_disease(C,D), disease_cause_symptom(D,anxiety). what_be_drug(C) :- drug_treat_disease(C,D), disease_be_related_to_gene(D,htr1a).
What are the diseases that cause Insomnia and are treated by a drug X? /*comment: ACE supports the use of variables. Here X is such a variable */	what_be_disease(C) :- disease_cause_symptom(C,insomnia), drug_treat_disease(E,C).

ACE Query	**Answer Set Program**
What are the symptoms of the diseases that are treated by a drug which is related to ADRB1 and which is not related to HTR1A?	what_be_symptom(C) :- symptom_of_disease(C,D), drug_treat_disease(E,D), drug_be_related_to_gene(E,adrb1), not drug_be_related_to_gene(E,htr1a).
What are the symptoms of the disease which is not caused by a gene which is not related to ADRB1? /*comment: negation inside negation is handled */	what_be_symptom(C) :- symptom_of_disease(C,D), not gene_cause_disease(F,D), gene_be_related_to_gene(F,adrb1).