AD-CASPAR: Abductive-Deductive Cognitive Architecture based on Natural Language and First Order Logic Reasoning

Carmelo Fabio Longo and Corrado Santoro

Department of Mathematics and Computer Science, University of Catania, Viale Andrea Doria, 6, 95125 Catania, Italy
fabio.longo@unict.it, santoro@dmi.unict.it

Abstract. In this paper, a Cognitive Architecture leveraging Natural Language Processing and First Order Logic Inference is presented, making usage of different kinds of knowledge bases interacting one another. Such a framework is able to make reasoning on queries requiring also combinations of axioms, represented by means of a rich semantic, using Abduction as pre-stage of Deduction. As application test a Telegram chatbot system has been implemented, supported by a module which automatically transforms polar and wh-questions into one or more likely assertions, in order to infer boolean values or snipplets with variable length as factoid answer. Furthermore, such a chatbot does not need script updates or code refactory when new knowledge has to income, but just the knowledge itself in natural language.

Keywords: Cognitive Architectures · Chatbots · Natural Language Processing · First Order Logic

1 Introduction

Among the applications leveraging Natural Language Processing (NLP), those related to Chatbots systems are growing very fast and present a wide range of choices depending on the usage, each with different complexity levels, expressive powers and integration capabilities. The first distinction between the chatbot platforms divides them into two big macro-categories: goal-oriented and conversational. The former is the most frequent kind, often designed for business platforms support, assisting users on tasks like buying goods or execute commands in domotic environments. In this case, it is crucial to extract from a utterance the intentions together with the related parameters, then to execute the wanted operation, providing then a proper feedback to the user. As for conversational ones, they are mainly focused on having a conversation, giving the

user the feeling to communicate with a sentient being, returning back reasonable answers optionally taking into account discussions topics and past interactions. The early shared aim for conversational chatbot systems was to pass the Turing test, hence to fool the user about his interlocutor; the state-of-art of such chatbot systems can be probed in the scope of the Loebner Prize Competition [1].

One of the most common platforms for building conversational chatbot is AIML [2] (Artificial Intelligence Markup Language), based on words pattern-matching defined at design-time; in the last decade it has become a standard for its flexibility to create conversation. In [3] AIML and Chatscript [4] are compared and mentioned as the two widespread opensource frameworks for building chatbots. On the other hand, AIML chatbots are difficult to scale if patterns are manually built, they have great limitations on information extraction capabilities and they are not suitable for task oriented chatbots. Other kinds of chatbots are based on deep learning techniques [5], making usage of huge corpus of examples of conversations to train a generative model that, given an input, is able to generate the answer. In general, all chatbots are not easily scalable without writing additional code or retrain a model with fresh datasets.

In this work, we present a cognitive architecture called AD-CASPAR based on NLP and First Order Logic (FOL) Reasoning, as baseline platform for implementing scalable and flexible chatbots with both goal-oriented and conversational features; nevertheless, this architecture leverages Question Answering techniques and is able of combining facts and rules in order to infer new knowledge from its own Knowledge Base. This first prototype is not yet capable of implementing a chatbot with complex dialog system, but differently from other platforms, in order to handle additional question-answer couples, the user has to provide just the related sentences in natural language. After the agent has parsed every sentence, a FOL representation is asserted in the Knowledge Base; in such a process, as we will show in the next sections, the Knowledge Base is able to act as a deductive database [6]. The basic idea is that, for dealing with new sentences, there is no need of updating the chatbot code at design-time. For achieving snipplets of knowledge (or a custom text) after a question, you have to give just the related knowledge before.

AD-CASPAR inherits most of its features directly from its predecessor CASPAR[7], whose name stands for: *Cognitive Architecture System Planned and Reactive*. The latter was designed to build goal-oriented agents (vocal assistants) with enhanced deductive capabilites, working on domotic environments; in the Github[1] repository all its features and information representations are shown in detail. The additional features introduced in AD-CASPAR are the usage of Abduction as pre-stage of the Deduction (that's why the presence of AD before CASPAR), in order to make inference only on a narrow set of query-related clauses, plus the application of Question-Answering techniques to deal with wh-questions and give back factoid answers (single nouns or snipplets) in the best cases; otherwise, optionally, only a relevance-based output will be returned.

[1] http://www.github.com/fabiuslongo/pycaspar

This paper is structured as follows: Section 2 shows in detail all the architecture's components and underlying modules; Section 3 shows how AD-CASPAR deals with polar and wh-questions; Section 4 summarizes the content of the paper and provides our conclusions, together with future work perspectives. A Python prototype implementation of AD-CASPAR is also provided for research purposes in a Github repository[2].

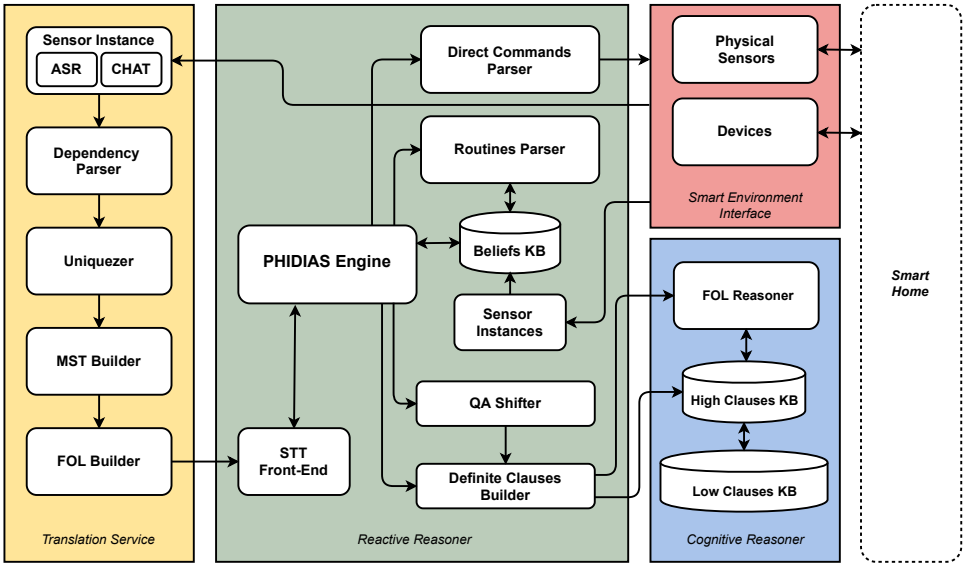

Fig. 1. The Software Architecture of AD-CASPAR

2 The Architecture

The main component of this architecture, namely the *Reactive Reasoner* (central box in Fig.1), acts as "core router" by delegating operations to other components, and providing all needed functions to make the whole system fully operative.

The Knowledge Base (KB) is divided into two distinct parts operating separately, which we will distinguish as *Beliefs KB* and *Clauses KB*: the former contains information of physical entities which affect the agent and which we want the agent to affect; the latter contains conceptual information not perceived by agent's sensors, but on which we want the agent to make logical inference. Moreover, the Clauses KB is divided into two different layers: *High Clause KB* and *Low Clauses Kb* (bottom right box in Fig.1). The totality of the knowledge is stored in the low layer, but the logical inference is achieved in the high one,

[2] http://www.github.com/fabiuslongo/ad-caspar

whose clauses will be the most relevant for the query in exam, taking in account of a specific confidence threshold which will be discussed later.

The Beliefs KB provides exhaustive cognition about what the agent could expect as input data coming from the outside world; as the name suggests, this cognition is managed by means of proper beliefs that can - in turn - activate proper plans in the agent's behaviour.

The Clauses KB is defined by the means of assertions/retraction of *nested* First Order Logic (FOL) definite clauses, which are possibly made of composite predicates, and it can be interrogated providing answer to any query (*True* or *False*).

The two KBs represent, somehow, two different types of human being memory: the so called *procedural memory* or *implicit memory*[8], made of thoughts directly linked to concrete and physical entities; the *conceptual memory*, based on cognitive processes of comparative evaluation. Nevertheless, the two layers of the Clauses KB can be seen as *Short Term Memory* (High Clauses KB) and *Long Term Memory* (Low Clauses KB).

As well as in human being, in this architecture, Belief KB and Clauses KB can interact with each other in a very reactive decision-making process.

2.1 The Translation Service

This component (left box in Fig. 1) is a pipeline of five modules with the task of taking an utterance in natural language and translating it in a *neo-davidsonian* FOL expression inheriting the shape from the event-based formal representation of Davidson [9], where the sentence for instance:

$$\textit{Brutus stabbed suddenly Caesar in the agora} \tag{1}$$

is represented by the following notation:

$$\exists \texttt{e stabbed(e, Brutus, Caesar)} \wedge \texttt{suddenly(e)} \wedge \texttt{in(e, agora)} \tag{2}$$

where the variable `e`, which we define as *davidsonian* variable, identifies the verbal action related to *stabbed*. In the case a sentence contains more than one verbal phrase, we'll make usage of indexes for distinguish e_i from e_j with $i \neq j$. As for the predicates arguments, in order to permit the sharing of qualitative features between predicates, whether we include (for instance) the adjective *evil* related to *Brutus*, the 2 can be changed as it follows:

$$\exists \texttt{e stabbed(e, Brutus(x), Caesar(y))} \wedge \texttt{evil(x)} \wedge \texttt{suddenly(e)} \wedge \texttt{in(e, agora(z))}$$

Furthermore, in the notation used for this work each predicate label is in the form `L:POS(t)`, where `L` is a lemmatized word and `POS` is a Part-of-Speech tag from the Penn Treebank [10] tagset.

The first module in the pipeline, namely *Sensor Instance*, can include either a module of Automatic Speech Recognition (ASR) or a module getting plain

text from a chatbot environment; the former allows a machine to understand the user's speech and convert it into a series of words.

The second module is the *Dependency Parser*, which aims at extracting the semantic relationships, namely *dependencies*, between all words in a utterance. All the dependencies used in this paper are part of the ClearNLP[11] tagset, which is made of 46 distinct entries.

The third module, the *Uniquezer*, aims at renaming all the entities within each dependency taking in account of the words offset, in order to make them unique. Such a task is mandatory to ensure the correctness of the outcomes of the next module in the pipeline (the *Macro Semantic Table*), whose data structures need a distinct reference to each entity coming from the dependency parser.

The fourth module, defined as *MST Builder*, is made of production rules leveraging semantic dependencies, with the purpose of building a novel semantic structure defined as *Macro Semantic Table* (MST). The latter summarizes in a canonical shape all semantic features in a sentence, in order to derive FOL expressions. Here is a general schema of a MST, referred to the utterance u:

$$\texttt{MST(u)} = \{\texttt{ACTIONS, VARLIST, PREPS, BINDS, COMPS, CONDS}\}$$

where

$$\begin{aligned}
\texttt{ACTIONS} &= [(\texttt{label}_k, \texttt{e}_k, \texttt{x}_i, \texttt{x}_j), \ldots] \\
\texttt{VARLIST} &= [(\texttt{x}_1, \texttt{label}_1), \ldots (\texttt{x}_n, \texttt{label}_n)] \\
\texttt{PREPS} &= [(\texttt{label}_j, (\texttt{e}_k \mid \texttt{x}_i), \texttt{x}_j), \ldots] \\
\texttt{BINDS} &= [(\texttt{label}_i, \texttt{label}_j), \ldots] \\
\texttt{COMPS} &= [(\texttt{label}_i, \texttt{label}_j), \ldots] \\
\texttt{CONDS} &= [\texttt{e}_1, \texttt{e}_2, \ldots]
\end{aligned}$$

All tuples inside such lists are populated with variables and labels whose indexing is considered disjoint among distinct lists, although there are significant relations which will be clarified shortly. The MST building takes into account also the analysis done in [12] about the so-called *slot allocation*, which indicates specific policies about entity's location inside each predicate, depending on verbal cases. This is because the human mind, in the presence of whatever utterance, is able to populate implicitly any semantic role (subject and object) taking part in a verbal action, in order to create and interact with a logical model of the utterance. In this work, by leveraging a step-by-step dependencies analysis, we want to create artificially such a model, to give an agent the chance to make logical inference on the available knowledge. For instance, considering the dependencies of 1:

```
nsubj(stabbed, Brutus)
ROOT(stabbed, stabbed)
advmod(stabbed, suddenly)
dobj(stabbed, Caesar)
prep(stabbed, In)
det(agora, The)
pobj(in, agora)
```

from the couple `nsubj/dobj` it is possible to create a new tuple inside `ACTIONS` as it follows, taking also in account of variables indexing counting:

$$\texttt{(stabbed, } e_1\texttt{, } x_1\texttt{, } x_2\texttt{)}$$

and inside `VARLIST` as well:

$$\texttt{(}x_1\texttt{, Brutus), (}x_2\texttt{, Caesar)}$$

Similarly, after an analysis of the couple `prep/pobj` it is possibile to create further tuples inside `PREPS` and `VARLIST` like it follows, respectively:

$$\texttt{(in, } e_1\texttt{, } x_3\texttt{), (}x_3\texttt{, agora)}$$

The dependency `advmod` contains informations about the verb (*stabbed*) is going to modify by means the adverb *suddenly*. In light of this, the tuple $\texttt{(}e_1\texttt{, suddenly)}$ will be created inside `VARLIST`.

As for the `BINDS` list, it contains tuples with a quality-modifier role: in the case the 1 had *the brave Caesar* as object, considering the dependency `amod(Caesar, brave)` a *bind* `(Caesar, brave)` will be created inside `BINDS`.

As with `BINDS`, `COMPS` contains tuples of terms related to each other, but in this case they are part of multi-word nouns like *Barack Hussein Obama*, which will be classified with the `compound` dependency.

The `CONDS` lists contains davidsonian variables whose related tuples within the `MST` subordinate the remaining others. For instance, in the presence of utterances like:

if the sun shines strongly, Robert drinks wine

or

while the sun shines strongly, Helen smiles

in both cases, the dependencies `mark(shines, If)`, `mark(shines, while)` will give informations about subordinate conditions related to the verb *shines*; in those cases, the davidsonian variable related to *shines* will populate the list `CONDS`. In the same way, in presence of the word *when* a subordinate condition might be inferred as well: since it is classified as `advmod` like whatever adverb, it might be considered as subordinate condition only when its POS is `WRB` and not `RB`, where the former denotes a wh-adverb and the latter a qualitative adverb. Unfortunately, such POS-based distinction is not sufficient, since also the adverb *where* is classified in the same way, which is indicative of a *location* where conditions related to some verbal action take place. So, depending from the domain, for achieving a comprehensive strategy in such a direction, a grammatical analysis is also required.

The fifth and last module, defined as *FOL Builder*, aims to build FOL expressions starting from the MSTs. Since (virtually) all approaches to formal

semantics assume the Principle of Compositionality[3], formally formulated by Partee [13], every semantic representation can be incrementally built up when constituents are put together during parsing. In light of the above, it is possible to build FOL expressions straightforwardly starting from a MST, which is built in a step-by-step semantic dependencies analysis. For instance, considering the sentence:

When the sun shines strongly, Robert is happy (3)

As effect of the Uniquezer processing before the MST building, which concatenate to each lemma its indexing in the body of the sentence among more occurrency of the same word, the related MST is:

$$\begin{aligned} \texttt{ACTIONS} &= [(\texttt{shine01:VBZ}, e_1, x_1, x_2), \\ &\quad \texttt{be01:VBZ}(e_2, x_3, x_4)] \\ \texttt{VARLIST} &= [(x_1, \texttt{sun01:NN}), (x_2, ?), (x_3, \texttt{Robert01:NNP}), (x_4, \\ &\quad \texttt{happy01:JJ})] \\ \texttt{CONDS} &= [e_1] \end{aligned}$$

The final outcome will be an implication like the following:

$$\texttt{shine01:VBZ}(e_1, x_1, \_) \wedge \texttt{sun01:NN}(x_1) \implies \texttt{be01:VBZ}(e_2, x_3, x_4) \wedge \texttt{Robert01:NNP}(x_3) \wedge \texttt{happy01:JJ}(x_4)$$

Since the MST Builder is made of production rules whom takes in account of relations (dependencies) between words, as long as such such relations are treated properly, the accuracy of the conversion from natural language can be considered equal to the accuracy of the dependency parser.

In order to obtain a disambiguation between words as well, which will be reflected on the predicate's labels, a naive strategy (inherited from CASPAR) is to possibly exploiting the doc2vect [14] similarity between the sentence containing the lemma and the WordNet examples (whether existing) or glosses defined within the synsets including such a lemma. The code of the most likely synset whose example similiarity is greater, will be choosen as part of the predicate's label.

2.2 The Reactive Reasoner

As already mentioned, this component (central box in Fig. 1) has the task of letting other modules communicate with each other; it also includes additional modules such as the Speech-To-Text (SST) Front-End, IoT Parsers (Direct Command and Routines), Sensor Instances, and Definite Clauses Builder. The Reactive Reasoner contains also the Beliefs KB, which supports both Reactive and Cognitive Reasoning.

The core of this component processing is managed by the BDI framework Phidias [15], which gives Python programs the ability to perform logic-based

[3] "The meaning of a whole is a function of the meanings of the parts and of the way they are syntactically combined."

reasoning (in Prolog style) and lets developers write reactive procedures, i.e., pieces of program that can promptly respond to environment events.

The agent's first interaction with the outer world happens through the STT Front-End, which is made of production rules reacting on the basis of specific beliefs asserted by a Sensor Instance; the latter, being instance of the superclass *Sensor* provided by Phidias, will assert a belief called `STT(X)` with `X` as the recognized utterance, after the sound stream is acquired by a microphone and translated by the ASR or acquired from a chatbot environment.

The Direct Command and Routine Parsers have the task of combining FOL expressions predicates with common variables coming from the Translation Services, via a production rules system. The former produces beliefs which might trigger operation executions, while the latter produces *pending* beliefs which need specific conditions before being treated as direct commands.

The Definite Clauses Builder is responsible of combining FOL expression predicates with common variables, through a production rules system, in order to produce nested definite clauses. Considering the 3 and its related FOL expression producted by the Translation Service, the Definite Clauses Builder, taking in account of the Part-of-Speech of each predicate, will produce the following *nested* definite clause:

$$\texttt{shine01:VBZ(sun01:NN(x}_1\texttt{), \_)} \implies \texttt{be01:VBZ(Robert01:NNP(x}_3\texttt{),}\ \texttt{happy01:JJ(}x_4\texttt{))}$$

The rationale behind such a notation choice is explained next: a definite clause is either atomic or an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. Because of such restrictions, in order to make MST derived clauses suitable for doing inference with the Backward-Chaining algorithm (which requires a KB made of definite clauses), we must be able to incapsulate all their informations properly. The strategy followed is to create composite terms, taking into account of the Part-of-Speech tags and applying the following hierarchy to every noun expression as it follows:

$$\texttt{IN(JJ(NN(NNP(x))), t)} \tag{4}$$

where `IN` is a preposition label, `JJ` an adjective label, `NP` and `NNP` are noun and proper noun labels, x is a bound variable and t a predicate.
As for the verbal actions, the nesting hierarchy will be the following:

$$\texttt{ADV(IN(VB(t}_1\texttt{, t}_2\texttt{), t}_3\texttt{))}$$

where `ADV` is an adverb label, `IN` a preposition label, `VB` a verb label, and t_1, t_2, t_3 are predicates; in the case of imperative or intransitive verb, instead of respectively t_1 or t_2, the arguments of `VB` will be left void. As we can see, a preposition (`IN`) might be related either to a noun or a verb.

2.3 The Smart Environment Interface

This component (upper right box in Fig.1) provides a bidirectional interaction between the architecture and the outer world. A production rules system is used

as reactive tool to trigger proper plans in the presence of specific beliefs. In [16] we have shown the effectiveness of this approach by leveraging the Phidias predecessor Profeta[17], even with a shallower analysis of the semantic dependecies, as well as an operations encoding via WordNet[18] in order to make the operating agent multi-language and multi-synonimous.

2.4 The Cognitive Reasoner

This component (right bottom box in Figure 1) allows an agent to assert/query the Clauses KB with *nested* definite clauses, where each predicate argument can be another predicate and so on, built by the Definite Clauses Builder module as shown in 2.2.

Beyond the nominal FOL reasoning with the known Backward-Chaining algorithm, this module exploits also another class of logical axioms entailed from the Clauses KB: the so-called *assignment rules*. We refer to a class of rules of the type "P is-a Q" where P is a predicate whose variable travels across one hand-side to another of an implicative formula, as argument of another predicate Q. For example, if we want to express the concept: *Robert is a man*, we can use the following closed formula:

$$\forall \texttt{x Robert(x)} \implies \texttt{man(x)} \tag{5}$$

But before that, we must consider a premise: if predicates are built from semantic dependencies, the introduction of such rules in a KB can be possible only by shifting from a strictly semantic domain to a pure conceptual one, because in a semantic domain we have just the knowledge of morphological relationships between words given by their syntactic properties. Basically, we need a medium to give additional meaning to our predicates which is provided by WordNet [18]. This allows us to make logical reasoning in a conceptual space thanks to the following functions:

$$F_I : P_S \longrightarrow P_C \qquad F_{Args(F_I)} : X_S^n \longrightarrow Y_C^n \tag{6}$$

F_I is the *Interpreter Function* between the space of all semantic predicates which can be yield by the `MST` sets and the space of all conceptual predicates P_C having a synset as label; it is not injective, because a single semantic predicate might have multiple corrispondences in the codomain, one for each different synset containing the lemma in exam. $\mathrm{F}_{Args(F_I)}$ is between domain and codomain (both with arity equal to n) of all predicate's argument of F_I. For instance, considering the MST derived FOL expression of *Robert is a man*:

$$\texttt{be:VBZ(e}_1\texttt{, x}_1\texttt{, } x_2\texttt{)} \wedge \texttt{Robert:NNP(x}_1\texttt{)} \wedge \texttt{man:NN(}x_2\texttt{)}$$

After an analysis of *be*, we find the lemma within the WordNet synset encoded by `be.v.01` and defined by the gloss: *have the quality of being something*. This is the medium we need for the domain shifting which gives a common sense meaning to our predicates.

In light of above, in the new conceptual domain given by (6), the same expression can be rewritten as:

$$\texttt{be.v.01\_VBZ}(d_1, y_1, y_2) \wedge \texttt{Robert\_NNP}(y_1) \wedge \texttt{man.n.01\_NN}(y_2)$$

where VBZ indicates the present tense of be.v.01, Robert_NNP(x) means that x identify the person *Robert*, and man.n.01_NN(x) means that x identify *an adult person who is male (as opposed to a woman)*.

Considering the meaning of be.v.01_VBZ, it does make sense also to rewrite the formula as:

$$\forall y\ \texttt{Robert\_NNP}(y) \implies \texttt{man.n.01\_NN}(y) \tag{7}$$

where y is a bound variable like x in (5).

Having such a rule in a KB means that we can implicitly admit additional clauses having man.n.01_NN(y) as argument instead of Robert_NNP(y).

The same expression, of course, in a conceptual domain can also be rewritten as a composite fact, where Robert_NNP(y) becomes argument of man.n.01_NN(y) as it follows:

$$\texttt{man.n.01\_NN}(\texttt{Robert\_NNP}(y)) \tag{8}$$

which agrees with the hierarchy of 4 as outcome of the Definite Clauses Builder.

As claimed in [19], not every KB can be converted into a set of definite clauses because of the single-positive-literal restriction, but many KB can, like the one related to this work for the following reasons:

1. No clauses made of one single literal will ever be negative, due to the closed world assumption. Negations, initially treated like whatever adverb, when detected and related to *ROOT* dependency are considered as polarity inverter of verbal phrases; so, in this case, any *assert* will be turned into a *retract*.
2. When the right hand-side of a clause is made by more than one literals, it is easy to demonstrate that, by applying the implication elimination rule and the principle of distributivity of $\vee$ over $\wedge$, a non-definite clause can be splitted into n definite clauses (where n is the number of consequent literals).

3 Question Answering

In this section is shown how this architecture deals with Question-Answering. Differently from its predecessor CASPAR, which works with a single/volatile Clauses KB, AD-CASPAR can count on a two-layer Clauses KB: High Clauses KB and Low Clauses KB. Every assertion is made on both the layers, but the logical inference is made only on the High one. As for the queries, whether a reasoning fails, the Low Clauses KB is used to populate the High one with relevance-based clauses, taking in account of the presence of common features between the clause-query and the clauses stored in the Low Clauses KB. Each record in the Low Clauses KB is stored in a NoSQL database and is made of three fields: Nested Definite Clause, Features Vector and the sentence in natural language. The Features Vector is made of all the labels composing the clause. For instance let the sentence to be stored be:

Barack Obama became president of United States in 2009

In this case, the record stored in the Low Clauses KB will be as it follow[4]:

- `In_IN(Become_VBD(Barack_NNP_Obama_NNP(x1), Of_IN(President_NN(x2), United_NNP_States_NNP(x3))), N2009_CD(x4))`
- `[In_IN, Become_VBD, Barack_NNP_Obama_NNP, Of_IN, President_NN, United_NNP_States_NNP, N2009_CD]`
- `Barack Obama became president of United States in 2009`

The abductive strategy of transfer from Low Clauses KB to High Clauses KB takes in account of a metric defined $Confidence_c$ as it follows, between a records in the Low Clauses KB and the query:

$$Confidence_c = \frac{|\bigcap(Feats_q, Feats_c)|}{|Feats_q|} \tag{9}$$

where $Feats_q$ is the Features Vector extracted from the query, and $Feats_c$ is the Features Vector in a record stored in the Low Clauses KB.
Once obtained the sorted list of all Confidences, together with the related clauses, the two most relevant clauses will be copied in the High Clauses KB. Such an operation is accomplished in a fast and efficient way by leveraging NoSQL collections indexes and the function *aggregation*[5] of MongoDB. The threshold Confidence of the clauses admitted to populate the High Clauses KB, can be defined at design time by changing a proper parameter in the file `config.ini` of the Github repository; of course, the more high the Confidence threshold the more relevant to the query will be the clauses transferred from the Low Clauses KB to the High one.

3.1 Polar Questions

Polar questions in the form of nominal assertion (excepting for the question mark at the end) are transformed in definite clauses and treated as query as they are, while those beginning with an auxiliary term, for instance:

Has Margot said the truth about her life?

can be distinguished by means the dependency `aux(said, Has)` and they will be treated by removing the auxiliary and considering the remaining text (without the ending question mark) as source to be converted into a clause-query.

[4] Supposing all predicates labels properly choosen among all synsets.
[5] For further details we remind the reader to inspect the file lkb_manager.py in the Github repository.

3.2 Wh-Questions

Differently from polar questions, for dealing with wh-question we have to transform the question into assertions one can expect as likely answer. To achieve that, after an analysis of several types of questions for each category[6], by leveraging the dependencies of the questions, we found it useful to divide the sentences text into specific chunks as it follows:

`[PRE_AUX] [AUX] [POST_AUX] [ROOT] [POST_ROOT] [COMPL_ROOT]`

The delimiter indexes between every chunk are given by `AUX` and `ROOT` related words positions in the sentence. The remaining chunks are extracted on the basis of the latters. For the likely answers composition, the module `QA Shifter` has the task of recombining the question chunks in a proper order, considering also the type of wh-question. Such a operation, which is strictly language specific, is accomplished thanks to an ad-hoc production rule system. For instance, let the question be:

Who could be the president of America?

In this case, the chunks sequence will be as it follows:

`[PRE_AUX] [could] [POST_AUX] [be] [the president of America] [COMPL_ROOT]`

where only the AUX, ROOT and POST_ROOT chunks are populated, while the others are empty. In this case a specific production rule of the QA Shifter will recombine the chunks in a different sequence, by adding also another specific word, in order to compose a proper likely assertion like it follow:

`[PRE_AUX] [POST_AUX] [the president of America] [could] [be] [COMPL_ROOT] [Dummy]`

At this point, joining all the words in such a sequence, the likely assertion to use as query will be the following:

The president of America could be Dummy

The meaning of the keyword *Dummy* will be discussed next. In all verbal phrases where `ROOT` is a copular verb[7] (like *be*), i.e., a non-transitive verb but identifying the subject with the object (in the scope of a verbal phrases), the following sequence will also be considered as likely assertion.

Dummy could be the president of America

[6] Who, What, Where, When, How

[7] The verbs for which we want to have such a behaviour can be defined by a parameter in a configuration file. For further details we refer the reader to the documentation in this work's Github repository.

All wh-questions for their own nature require a *factoid* answer, made of one or more words (snipplet); so, in the presence of the question: *Who is Biden?* as answer we expect something like: *Biden is Something*. But *Something* surely is not what we are looking for as information, but *the elected president of United States* or something else. This means that, within the FOL expression of the query, *Something* must be represented by a mere variable and not a ground term. In light of this, instead of *Something*, this architecture uses the keyword *Dummy*; during the creation of a FOL expression containing such a word, the Translation Service will impose the Part-of-Speech `DM` to *Dummy*, whose parsing is not expected by the Clauses Builder, thus it will discarded. At the end of this process, as FOL expression of the query we'll have the following literal:

```
Be_VBZ(Biden_NNP(x1), x2)
```
(10)

which means that, if the High Clauses KB contains the representation of *Biden is the president of America*, namely:

```
Be_VBZ(Biden_NNP(x1), Of_IN(President_NN(x2), America_NNP(x3)))
```

querying with the 10 by using the Backward-Chaining algorithm, as result it will return back a unifying substitution with the previous clause as it follows:

```
{v_41: x1, x2: Of_IN(President_NN(v_42), America_NNP(v_43))}
```
(11)

which contains, in correspondence of the variable `x2`, the logic representation of the snipplet: *president of America* as possible and correct answer. Furthermore, starting form the lemmas composing the only answer-literal within the substitution, with a simple operation on a string it is possible to obtain the minimum snipplet of the original sentence containing such lemmas.

4 Conclusions and Future Works

In this paper we have presented a Cognitive Architecture called AD-CASPAR, based on Natural Language Processing and FOL Reasoning, capable of Abductive Reasoning as pre-stage of Deduction. By the means of its module Translation Service, it parses sentences in natural language in order to populate its KBs with beliefs or nested definite clauses using a rich semantic. Moreover, the module QA Shifter is able to rephrase wh-questions into likely assertions one can expect as answer, thanks to a production rule system which leverages also a dependency parser. The combination of Translation Service/Definite Clause Builder and QA Shifter makes the Telegram Bot proposed in this work easily scalable on the knowledge we want it to deals with, because the user has to provide just the new sentences in natural language at runtime, like in a normal conversation.

As future work, we want to include a module for the design of Dialog Systems, taking in account also of contexts and history. Furthermore, we want to exploit external ontologies for getting richer answers, and to design an additional module inspired to the human hippocampus, to let the agent spontaneously link together knowledge for relevance in order to enhance the Dialog System.

References

1. H. Loebner, "The loebner prize." Available at https://www.ocf.berkeley.edu/ arihuang/academic/research/loebner.html.
2. A. fondation, "Artificial intelligence markup language." Available at http://www.aiml.foundation/.
3. H. Madhumitha.S, Keerthana.B, "Interactive chatbot using aiml," *Int. Jnl. Of Advanced Networking and Applications*, vol. Special Issue, 2019.
4. B. Wilcox, "Chatscript." Available at https://github.com/ChatScript/ChatScript.
5. Q. V. L. Ilya Sutskever, Oriol Vinyals, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 27, 2014.
6. J. H. Kotagiri Ramamohanarao, "An introduction to deductive database languages and systems," *The International Journal of Very Large Data Bases*, vol. Journal, 3, 107-122, 1994.
7. C. S. Carmelo Fabio Longo, Francesco Longo, "A reactive cognitive architecture based on natural language processing for the task of decision-making using a rich semantic," in *21st Workshop "From Objects to Agents" (WOA 2020)*, 2020.
8. D. Schacter, "Implicit memory: history and current status," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. vol. 13, 1987, pp. 501–518, 1987.
9. D. Davidson, "The logical form of action sentences," in *The logic of decision and action*, p. 81–95, University of Pittsburg Press, 1967.
10. L. D. Consortium, "Treebank-3." Available at https://catalog.ldc.upenn.edu/LDC99T42.
11. ClearNLP, "Clear nlp tagset." Available at https://github.com/clir/clearnlp-guidelines.
12. S. Anthony and J. Patrick, "Dependency based logical form transformations," in *SENSEVAL-3: Third International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*, 2015.
13. B. H. Partee, *Lexical Semantics and Compositionality*, vol. 1, p. 311–360. Lila R. Gleitman and Mark Liberman editors, 1995.
14. T. M. Quoc Le, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning, Beijing, China*, 2014.
15. C. S. Fabio D'Urso, Carmelo Fabio Longo, "Programming intelligent iot systems with a python-based declarative tool," in *The Workshops of the 18th International Conference of the Italian Association for Artificial Intelligence*, 2019.
16. C. F. Longo, C. Santoro, and F. F. Santoro, "Meaning Extraction in a Domotic Assistant Agent Interacting by means of Natural Language," in *28th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE, 2019.
17. L. Fichera, F. Messina, G. Pappalardo, and C. Santoro, "A python framework for programming autonomous robots using a declarative approach," *Sci. Comput. Program.*, vol. 139, pp. 36–55, 2017.
18. G. A. Miller, "Wordnet: A lexical database for english," in *Communications of the ACM Vol. 38, No. 11: 39-41*, 1995.
19. P. N. Stuart J. Russel, *Artificial Intelligence: A Modern Approach*, ch. 9.3. Pearson, 2010.