September 2018

Implementing the ArgQL Query Language

Dimitra ZOGRAFISTOU [a], Giorgos FLOURIS [a], Theodore PATKOS [a] and Dimitris PLEXOUSAKIS [a]

[a] *Institute of Computer Science, Foundation for Research and Technology Hellas, Heraklion, Greece*

Abstract. Exploration and information identification constitute challenging research problems, with important applications in sensemaking over structured argumentative dialogues. In this paper, we present the implementation ArgQL, a high-level declarative query language, designed for querying dialogical data, structured in the principles of argumentation. We implement the language using an AIF-based representation and a translation of ArgQL into (complex) SPARQL queries. ArgQL provides a simple and intuitive way to query a structured dialogue using pure argumentative terminology.

Keywords. Argumentation, Declarative Query Languages, Graph data models, Semantic Web

1. Introduction

The recent advancements in the Web technologies transformed its users from passive information consumers to active creators of digital content. Web became a universal terrain, wherein humans accommodate their inherent need for communication and self-expression. This new era revealed several new research problems. Studying the informational requirements while navigating in dialogues and identifying specific pieces of data is one of the most challenging ones. The process of human argumentation on the other side has been an object of longstanding theoretical studies. Computational Argumentation [6,10] is a branch of AI and it serves theoretical and computational reasoning models that simulate human cognitive behavior while arguing [8,1,2]. It has been observed that it also defines solid and discrete constructs able to structure dialogical data.

Motivated by this observation, we introduced *ArgQL* (Argumentation Query Language) a high-level declarative query language, that allows for information extraction from a graph of structured and interconnected arguments. The initial specification of the language was presented in our previous work [13]. As highlighted there, ArgQL constitutes an effort to realize the requirements of querying argumentative data by translating them into particular lexicographic and syntactic rules of a language. Its syntax takes into account arguments' internal structure, as well as the abstract, graph-like view, shaped by the existing interrelations. It serves a simple and quite elegant mechanism to write queries in the argumentation domain like "How an argument with a given conclusion is attacked?". Its prominence is amplified by the expressive complexity of traditional lan-

guages (e.g. SPARQL), even for such simple statements. We discuss such issues and give illustrative examples in Section 3.

In this work, we describe our methodology and the technical details regarding the execution of ArgQL. In particular, since there is no native storage scheme that conforms to the principles of the suggested data model, we propose a mapping to the RDF data model and then we describe the translation method of ArgQL into the associated query language, SPARQL [9]. We choose to show the method in the context of the Semantic Web, however given that ArgQL has been designed as a domain independent language, we claim that it can be translated in any query language as long as there exist valid translation mechanisms.

The rest of the document is structured as follows. In Section 2 we give a brief description of the argumentation data model and ArgQL. Section 3 presents the mechanism for the query execution and we conclude in section 4, with a small discussion about the results so far and with some thoughts about future directions of this work.

2. Data model and Language

In this section we will give a brief description and some examples of the argumentation data model and ArgQL. Formal definitions and the language specification (syntax-semantics) are found in [13]. Due to the lack of space, we describe here informally some necessary concepts.

2.1. Argumentation Data model

The main concept in the data model is the *argument*. An argument is a tuple $\langle pr, c \rangle$, where pr is called *premise*, c is called *conclusion* and it holds that c is inferred by pr. Premise and conclusion are constituted by *propositions* taken from the infinite set $\mathcal{P}$. Arguments are organized in an argument base, which also contains relations among arguments, defined through *contrariness* and *equivalence*. In particular, *rebut* (mutually inconsistent conclusions) and *undercut* (conflict between one's conclusion and some premise of the other) are sub-relations of *attack*, while *endorse* (equivalent conclusions) and *backing* (equivalence between one's conclusion and some premise of the other) are sub-relations of *support*. Overall, the abstract view of the argument base forms a graph, whose nodes are structured arguments and in which, there are four different types of edges created by these four types of sub-relations. We also use the notion of *path* as a sequence of relations between two arguments in the argument graph.

2.2. ArgQL language

Let an infinite set V, which includes the variables of the language. Variables are prefixed with '?' and are used to bind with values from the data. Values of propositions are always in quotes " ". The general form of an ArgQL query is:

q ← ***match*** *dialogue_pattern (',' dialogue_pattern)**
return *varlist*

where *varlist* = $(v_1, v_2, \ldots, v_n)$, with $v_i \in V$ is a list of variables. A *dialogue pattern* may have one of the following two forms:

dialogue_pattern ::= argpattern | argpattern pathpattern dialogue_pattern

Argument patterns constitute primary units in the language and are designed to match the structure of arguments. Syntactically, an argument pattern can be either a single variable $v_a \in V$, or have the form v_a:⟨*premisePattern, conclusionPattern*⟩. *PremisePattern* and *conclusionPattern* restrict the premise and conclusion part of arguments, respectively. More precisely, the second form of an argument pattern may be one of the following:

$$v_a : \langle \{p_1, .., p_n\}, c \rangle \quad \text{or} \quad v_a : \langle v_p[f], c \rangle$$

where $p_i \in \mathcal{P}$, $c \in \mathcal{P} \cup V$, $v_p \in V$ and f a premise filter. Variable $\mathtt{v_p}$ matches the premise part of arguments and takes values from $2^{\mathcal{P}}$, whereas c matches the conclusion part and may be either a variable or a constant proposition value. The occurrence of the expression $[f]$ is optional. When existed, the premise is restricted based on a particular proposition set, let s, and we have 3 types of filters: *inclusion*, *join* and *disjointness* written as [/s] , [.s] and [!s], respectively. Below, we show some examples of argument patterns:

- ⟨?v[/{"p_1"}],?c⟩ : match arguments whose premise include some equivalent proposition of "p_1".
- ⟨?v,"c"⟩ : match arguments with conclusion any equivalent proposition of *"c"*
- ⟨?v[.{"p_1","p_2"}],?c⟩ : match arguments whose premise intersect with a set equivalent to {"p_1","p_2"}
- ⟨{"p_1","p_2"},"c"⟩ : instantiated arguments are also argument patterns

Path patterns are expressions that match with complete paths and allow for navigation in the graph of arguments. They are recognized by sequences of relations separated by the character '/' (e.g. `attack/support/support`). Note that a *relation* can be either one of the sub-relations (rebut, undercut, endorse, backing) or one of the general ones (attack, support). In the second case, it is indifferent which of the two sub-relations is satisfied. The expression **n* is a syntactic sugar to express the "*n* repetitions of a path pattern". For example `rebut*2`, is an alternative of `rebut/rebut`. In addition, we can express the case "up to *n* repetitions", by using the notation *'+n'*. In particular, `attack+3` defines three different path patterns: {`attack, attack/attack, attack/attack/attack`}. The coexistence of multiple number indicators in the same pattern gives a number of combinations, equal to the proliferation of the '+' indicators.

- ?a `(attack*2/support)+2` ⟨?pr_2,"c"⟩: matches with sequences of arguments, satisfying the path *attack/attack/support* or the *attack/attack/support/attack/attack/support* and also result to an argument with conclusion *"c"*.

Next, we show some examples of complete queries in ArgQL:

Q1. Find arguments which "defend"(attack the attackers of) arguments with conclusion equivalent to "Cloning is going to be awesome" and return all of them. Such queries are inspired by Dung [6] and can be used to implement some of his semantics.

```
match ?a1 (attack/attack)+3
?a2:<?pr, "Cloning is going to be awesome">
return ?a1, ?a2
```

Q2. Find and return arguments which attack arguments at distance of three support relations from an argument, which contains as one of its premises the proposition *"cloning contributes positively in artificial insemination"*, or an equivalent one.

```
match ?arg attack/(support)*3
<?pr[/{"cloning contributes ... insemination"}], ?c>
return ?arg, ?c
```

Q3. Find pairs of arguments whose premise sets intersect and return them.

```
match ?a1:<?pr1, ?c1>, ?a2:<?pr2[.?pr1], ?c2>
return ?a1, ?a2
```

3. Query execution

In this section, we describe the proposed implementation of ArgQL. Figure 1 depicts the execution flow for the translation into a arbitrary storage scheme. The *ArgQL Parser* part is responsible to recognize valid queries of the language. *Data and Query mappings* part bridges the gap between ArgQL and the actual data, by realizing the translation of queries and their results. Finally, part *KB* represents the data stored in a specific format (Relational, RDF, XML etc). Each different format should define different data and query mappings.

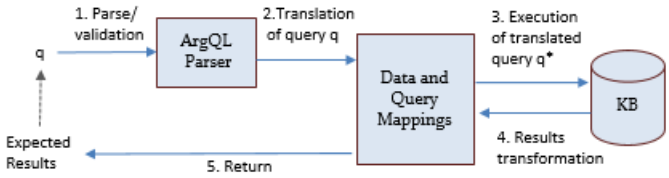

Figure 1. Process of query execution

In this work, we will show this flow for the RDF data model. To this end, we break down our strategy into the following tasks:

- Building an ArgQL parser
- Creating a dataset by defining an ontological scheme in RDFS and store RDF data of this form
- Formalizing the mappings between the concepts of the argumentation data model and the specific ontology
- Implementing a translation method of an ArgQL query into its respective SPARQL that will target the RDF data.

ArgQL parser. To implement the validation of ArgQL queries, we use the ANTLR compiler[1], a tool for language recognition, which provides a framework for constructing recognizers, compilers and translators from grammatical descriptions, containing Java, C++, or C# actions.

Ontology Scheme and Dataset description. In recent years, the community of Argument Web [3,7] seem to have consented into a standard RDF conceptualization for argumentative data, the AIF ontology [5,4]. In order to be compatible with this tendency, we also adopt that scheme to cover our requirements for this task. More precisely, we use its latest version, AIF+ [11], which constitutes an extension designed to capture concepts for dialogical argumentation. The core ontology of AIF constitutes a two-level ontology. The topmost layer consists of two disjoint sets of nodes, which are abstractly defined:

[1] http://www.antlr.org/

information nodes denoted by *I*, to hold the textual content, and *scheme nodes*, denoted by *S*. Scheme nodes are further specialized into three different and also disjoint sets of nodes for *inference application*, *preference application* and *conflict application*, denoted as *RA-, PA-, CA-node*, respectively. At this level, there are also various constraints on how components interact: information nodes, for example, can only be connected to other information nodes via scheme nodes of one sort or another. Scheme nodes, on the other hand, can be connected to other scheme nodes directly to express complex statements like conflicts on inferences, preferences on conflicts etc. At the lowest layer, these concepts become more specific by allowing via inheritence, to define for example particular inference schemes (e.g. presumptive), or represent the Walton schemes [12], retaining constraints defined in the layer above. AIF+ makes a separation between the representation of actual arguments and the representation of the normative structures in a dialogue protocol. Further classes are introduced at the dialogue side, like *L-node* called *locution node* that inherits *I-node*, *TA-node* called *transition node*, *YA-node*, called *anchor node* and *MA-node*, called *default rephrase*.

AIFdb corpora[2] is an online dataset that contains argumentative data structured in the AIF+. It contains several data formats, including RDF. Currently, it enumerates over 5000 argument maps, gathered by the various argument tools and with this number to be continuously increasing.

Data mapping. In this part, we will describe the mapping between the argumentation data model targeted by ArgQL and the concepts of AIF+. The following table includes this mapping. We use the notation *cls(x)* to denote the class instance in AIF, created by the concept *x* from the data model on the left. As a result, *cls* may have one of the types *(i, ra, ca, pa, loc, ya, ta, ma)* for the respective AIF+ classes I-, RA-, CA-, PA-, Locution-, YA-, TA-, MA-node. Each cell in the right column includes two sets: The set *nodes* describes the class instances in AIF to which the concept on the left cell is mapped. The set *edges* includes the edges between these instances. An edge is represented as $cls_1(x) - cls_2(y)$ (e.g. $i(p_1) - i(p_2)$ is an edge between two instances of the class I-node).

Argument data model	**AIF+**
Proposition p	nodes: $i(p)$, content of $i(p) = p$
Argument $a = \langle\{p_1, \ldots, p_n\}, c\rangle$	nodes: $i(p_1), \ldots, i(p_n), i(c), ra(a)$ edges: $i(p_1) - ra(a), \ldots, i(p_n) - ra(a), ra(a) - i(c)$
Conflict x between p_1 and p_2	nodes: $i(p_1), i(p_2), ca(x)$ edges: $i(p_1) - ca(x), ca(x) - i(p_2)$
Equivalence e between p_1 and p_2	nodes: $i(p_1), i(p_2), loc(p_1), loc(p_2), ya(p_1), ya(p_2), ta(e), ya(e), ma(e)$ edges: $i(p_1) - ya(p_1), i(p_2) - ya(p_2), ya(p_1) - loc(p_1), ya(p_2) - loc(p_2)$ $loc(p_1) - ta(e), ta(e) - loc(p_2), ta(e) - ya(e), ya(e) - ma(e)$

Table 1. Data mapping table

Each single proposition $p \in P$ generates a unique I-node, the information content of which is the value of p. Each argument a is mapped to an instance of *inference node* (RA), for which, there are n incoming edges from the I-nodes generated by the premises $p_1, \ldots, p_n$ and one outgoing edge to the I-node generated by the conclusion c. Conflicts

[2] http://www.corpora.aifdb.org/

between propositions p_1, p_2 are mapped to instance nodes of CA class, which lie between the I-nodes of p_1 and p_2. Finally, concerning the equivalences between propositions p_1, p_2, AIF+ introduces the notion of *default rephrase* represented by the class MA. To express the default rephrase between the I-nodes of p_1, p_2, a number of classes are created that make the transition to the dialogue side and a number of edges linking them that result to the particular instance of MA, as shown in the last row of the mapping table.

Query translation and execution. Due to the lack of space, we will give an visual description of the translation mechanism. For anyone interested to deepen in the formal details, there is a comprehensive analysis in the document [3].

The main idea is that each different element (usually pattern) of a query generates a small piece in the overall graph pattern. The final SPARQL query is built progressively, throughout the construction of the parse tree in the process of the lexicographic and syntactic recognition of q. During the whole process, new variables are created to bind with the URIs of the RDF resources and to perform the various joins. Each different alternative even in the same rule, usually gives a different result. Let the query:

```
q : match ?a1:⟨?pr1[/{"p1"}], ?c1⟩ attack ?a2:⟨?pr2, "c2"⟩,
    ?a3:⟨?pr3[/?pr1], ?c3⟩
    return ?a1, ?a2, ?a3
```

It asks to find arguments that attack those with conclusion "c_2", and then to find those, which have also as premises the premises of the "attackers".

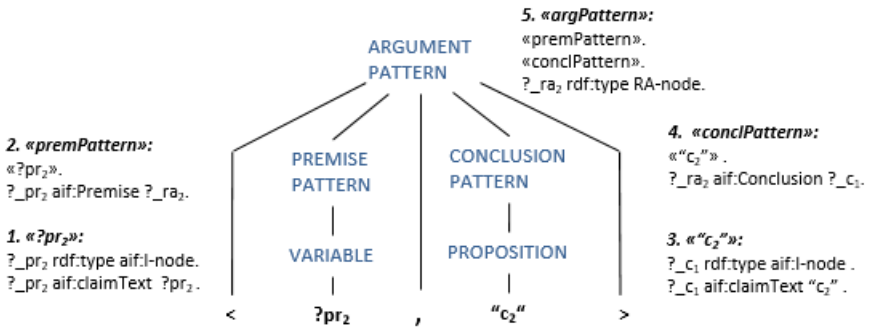

Figure 2. Part of the translation process of q

Figure 2 illustrates a part of the translation of q and in particular shows how the SPARQL graph pattern, that corresponds to the second argument pattern in q ($\langle ?pr_2, "c_2" \rangle$) is composed upon its parse tree generation. With $\langle\!\langle x \rangle\!\rangle$ we denote the part of graph pattern generated by the ArgQL component x. The leaves of the tree contain the tokens appearing in the ArgQL query, while the internal nodes represent the names of the grammar rules. In stages 1 and 2 the graph pattern for the premise pattern $?pr_2$ is created, denoted as $\langle\!\langle premPattern \rangle\!\rangle$. It is composed by the triple pattern *(?_pr2 aif:Premise ?_ra2)* and the graph pattern $\langle\!\langle ?pr_2 \rangle\!\rangle$, created by the invocation of the grammatical rule, in which a premise pattern is a single variable. Accordingly, the steps 3 and 4 show the creation of the graph pattern $\langle\!\langle conclPattern \rangle\!\rangle$ generated by the *conclusion pattern* "c_2". In the end,

[3] http://www.ics.forth.gr/isl/ArgQL/ArgQLtoSPARQL.pdf

at step 5 when the complete argument pattern has been recognized, the SPARQL graph pattern that corresponds to it is composed by the conjunction of the ⟪*premPattern*⟫ and ⟪*conclPattern*⟫ graph patterns and the triple pattern *(?_ra2 rdf:type aif:RA-node)*.

The next figure, shows the complete SPARQL generated by the query above. What is highlighted by this example is the expressive complexity of SPARQL to the particular requirements set by the ArgQL query. Apart from the difficulty to write such a query, it is also impossible for someone to read it and understand what it requires. The implementation of the language can be found here[4].

```
SELECT *
WHERE {
  ?_i1 rdf:type I-node.
  ?_i1 aif:claimText ?pr1 .
  ?_i1 aif:Premise ?_ra1.
  ?_i2 rdf:type I-node.
  ?_i2 aif:claimText "p1".
  { ?_i2 aif:Premise ?_ra1. }
  UNION
  {
   ?ya1 aif:IllocutionaryContent ?_i2.
   ?ya1 aif:Locution ?loc1.
   ?loc1 rdf:type aif:L-node.
   ?_i3 rdf:type aif:I-node.
   ?_i3 aif:claimText ?_i4.
   ?ya2 aif:IllocutionaryContent ?_i3.
   ?ya2 aif:Locution ?loc2.
   ?loc2 rdf:type aif:L-node.
   ?ta1 rdf:type aif:TA-node.
   ?loc1 aif:StartLocution ?ta1.
   ?ta1 aif:EndLocution ?loc2.
   ?ya3 aif:Anchor ?ta1.
   ?ya3 aif:IllocutionaryContent ?_ma1.
   ?_ma1 rdf:type aif:MA-node .
   ?_i3 aif:Premise ?_ra1.
  }
  ?_c1 rdf:type I-node.
  ?_c1 aif:claimText ?c1 .
  ?_a1 aif:Conclusion ?_c1 .
  ?_a1 rdf:type RA-node.
  ?_i4 rdf:type I-node.
  ?_i4 aif:claimText ?pr2 .
  ?_i4 aif:Premise ?_ra2.
  ?_c2 rdf:type I-node.
  ?_c2 aif:claimText "c2".
  { ?_ra2 aif:Conclusion ?_c2 . }
  UNION
  {
   ?ya4 aif:IllocutionaryContent ?_c2.
   ?ya4 aif:Locution ?loc3.
   ?loc3 rdf:type aif:L-node.
   ?_i5 rdf:type aif:I-node.
   ?_i5 aif:claimText ?_i6.
   ?ya5 aif:IllocutionaryContent ?_i5.
   ?ya5 aif:Locution ?loc4.
   ?loc4 rdf:type aif:L-node.
   ?ta2 rdf:type aif:TA-node.
   ?loc4 aif:StartLocution ?ta2.
   ?ta2 aif:EndLocution ?loc4.
   ?ya5 aif:Anchor ?ta2.
   ?ya6 aif:IllocutionaryContent ?_ma2.
   ?_ma2 rdf:type aif:MA-node .
   ?_ra2 aif:Conclusion ?_i5.
  }
  ?_ra2 rdf:type RA-node.
  ?_ca rdf:type CA-node
  { ?_c1 aif:conflicting-element ?_ca.
   ?_ca aif:conflicted-element ?_c2. }
  UNION
  { ?_c1 aif:conflicting-element ?_ca.
   ?_ca aif:conflicted-element ?_i4 . }

  ?_i6 rdf:type I-node.
  ?_i6 aif:claimText ?pr3 .
  ?_i6 aif:Premise ?_ra3.
  FILTER NOT EXISTS {
    ?_x aif:Premise ?ra1.
    FILTER NOT EXISTS {
      ?_x aif:Premise ?ra3.
    }
  ?_c3 rdf:type I-node.
  ?_c3 aif:claimText ?c3.
  ?_ra3 aif:Conclusion ?_c3 .
  ?_ra3 rdf:type RA-node.
}
```

Figure 3. SPARQL query q*, generated by the translation of query q

4. Discussion and Conclusions

In this work, we presented the technical details concerning the implementation of ArgQL. The ability to execute ArgQL queries in real datasets, gives prominence also to its practical application and usefulness, except from the theoretic significance presented in [13]. To our point of view, ArgQL is a language with important prospects in the argumentation domain. It revealed several interesting issues relevant to the searching into dialogical data, the confrontation of which would enhance remarkably the process with more qualitative data extraction techniques. For example, the integration of the language with some reasoning mechanisms that will allow for dynamic computations and creation of data "at query time" is an idea. Such mechanisms will allow for Dung-style (and simi-

[4] http://www.ics.forth.gr/isl/ArgQL

lar) analysis of argumentation frameworks and the identification of acceptable arguments under various semantics. Furthermore, we plan to afford ”smart” searching mechanisms within the textual content of argument, such as advanced keyword-searching and content-based searching, imprecise textual mappings (e.g., taking into account synonyms, or typos in the text), exploratory/navigational capabilities etc.

References

[1] Trevor J. M. Bench-capon, Sylvie Doutre, and Paul E. Dunne. Value-based argumentation frameworks. In *Artificial Intelligence*, pages 444–453, 2002.

[2] Tudor Berariu. *An Argumentation Framework for BDI Agents*, pages 343–354. Springer International Publishing, Cham, 2014.

[3] Floris Bex, John Lawrence, Mark Snaith, and Chris Reed. Implementing the argument web. *Commun. ACM*, 56(10):66–73, October 2013.

[4] Floris Bex, Sanjay Modgil, Henry Prakken, and Chris Reed. On logical specifications of the argument interchange format. *Journal of Logic and Computation*, 23(5):951–989, 2012.

[5] Carlos Chesñevar, Jarred McGinnis, Sanjay Modgil, Iyad Rahwan, Chris Reed, Guillermo Simari, Matthew South, Gerard Vreeswijk, and Steven Willmott. Towards an argument interchange format. *Knowl. Eng. Rev.*, 21(4):293–316, December 2006.

[6] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, September 1995.

[7] Giorgos Flouris, Antonis Bikakis, Patkos Theodore, and Dimitris Plexousakis. Globally interconnecting persuasive arguments: The vision of the persuasive web. Technical report, FORTH-ICS/TR-438, 2013.

[8] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming: An argumentative approach. *Theory Pract. Log. Program.*, 4(2):95–138, January 2004.

[9] Steve H. Garlik, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/.

[10] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[11] Chris Reed, Simon Wells, Joseph Devereux, and Glenn Rowe. Aif+: Dialogue in the argument interchange format. In *COMMA*, 2008.

[12] Douglas N. Walton. *Argumentation Schemes for Presumptive Reasoning*. L. Erlbaum Associates, 1996.

[13] Dimitra Zografistou, Giorgos Flouris, and Dimitris Plexousakis. Argql: A declarative language for querying argumentative dialogues. In *International Joint Conference on Rules and Reasoning*, pages 230–237. Springer, 2017.