Extending the ArgQL Specification

Yannis Roussakis[1], Giorgos Flouris[1], Dimitra Zografistou[2] and Elisjana Ymeralli[1]

[1] *Institute of Computer Science, FORTH Heraklion, Crete, Greece*
[2] *Centre for Argument Technology (ARG-tech), University of Dundee*

Abstract.
Recent developments in Web technologies have transformed Web users from passive consumers to active creators of digital content. A significant portion of this content is of argumentative form, as users see the Web as a means to enable dialogical exchange, debating, and commenting on products, services or events. In this context, being able to identify, mine, represent, reason with, and query argumentative information found online is an important consideration. In previous work, some of the authors of this paper proposed ArgQL, a high-level declarative language for querying argumentative information found online. The current paper describes various extensions and improvements of ArgQL that bring it closer to actual use in realistic environments. These include methods to support more expressive keyword-based searching in arguments, and the support for querying non-argumentative information that is associated with arguments, such as the date of creation, author, topic etc (i.e., argument metadata).

Keywords
Computational argumentation, online debating, querying argumentative information, ArgQL, metadata, keyword search

1. Introduction

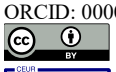
Recent advances in Web technologies transformed its users from passive information consumers to active creators of digital content. Web became a universal terrain, where humans accommodate their inherent need for communication and self-expression. This new era revealed several new research problems. Navigating in dialogues and identifying argumentative data is one of the most challenging ones. On the other hand, the process of human argumentation has been the object of study in Computational Argumentation [2, 4], a branch of AI that provides theoretical and computational reasoning models that simulate human cognitive behavior while arguing.

ArgQL (Argumentation Query Language) [9] is a high-level, representation-agnostic and declarative query language that allows for information extraction from a graph of structured and interconnected arguments (see Subsection 2.2). It allows accessing arguments stored in a repository, and is suitable for querying arguments in the Argument Web [5], through queries like "how an argument with conclusion X is attacked?". Such a repository could be created using a specialized tool for debate and argument generation (e.g., APOPSIS [8]), or through argument mining techniques from textual corpora.

In this paper, we improve ArgQL by proposing a set of extensions over its original specification (see Section 3). These extensions consist of the keyword search functionality over arguments (Subsection 3.1), as well as the introduction of a new notion, namely metadata, which is a versatile tool allowing the association of any property or path of properties with an argument, and the querying of arguments based on such metadata (Subsection 3.2). We argue that these functionalities allow for more meaningful queries, and constitute an important extension of the original specification.

This work was performed in the context of the DebateLab project[1], which conducts research towards developing the theoretical infrastructure for mining, representing and reasoning with online arguments,

[1] https://debatelab.ics.forth.gr/

RuleML+RR'22: 16th International Rule Challenge and 6th Doctoral Consortium, September 26--28, 2022, Virtual
EMAIL: rousakis@ics.forth.gr (A. 1); fgeo@ics.forth.gr (A. 2); dzografistou@gmail.com (A. 3); ymeralli@ics.forth.gr (A.4)
ORCID: 0000-0002-8937-4118 (A. 2)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

while delivering a suite of tools supporting the uptake of the related technologies in the domain of e-journalism.

The rest of the paper is structured as follows: we first give some preliminaries, including a short description of the original ArgQL (Section 2). In Section 3, we describe the functionality of the implemented extensions, their implementation, and how they can be used by the user. In Section 4, we describe how the ArgQL syntax was extended to support these additional features, and conclude in Section 5.

2. Preliminaries and Related Work

There are no equivalent languages to directly compare ArgQL with. Several tools have been developed to facilitate participation in online debates [7]. Although these tools allow graphical access to the provided arguments, none of them allows for a declarative query language for accessing arguments. In fact, the querying process internally employs traditional query languages such as SQL or SPARQL. ArgQL supports the different information needs of such tools and provides a language that allows the user to perform his/her own queries in a user-friendly manner.

2.1. AIF Ontology

The AIF ontology (Argument Interchange Format) [1] is a popular core ontology designed to represent arguments and their relations in a structured and systematic way. It is used as an abstract and high-level language that connects arguments from various argumentation tools and applications, and thus can be queryable and searchable by several search engines, such as ArgDF [3, 5], DiscourseDB2, and also ArgQL [9, 10]. The AIF specification3 is available in various formats[4], as described in [1]. An extension of AIF [6] provides better support for the representation of dialogues.

2.2. ArgQL Description

ArgQL [9, 11] is a high-level, representation-agnostic, declarative query language for argumentative information. Its syntax considers the arguments' internal structure, as well as an abstract, graph-like view of the dialogue, shaped by the existing interrelations among arguments. It allows the elegant formulation of queries on arguments and/or the associated dialogue. Its prominence is amplified by the fact that expressing the same information needs in traditional languages (e.g., SPARQL) would require the formulation of complex queries, even for simple statements. Moreover, to do so, one needs to be aware of the underlying representation scheme of arguments.

The syntax of ArgQL allows for expressions that filter the argumentative structure, combined with expressions used to identify sequences (paths) of arguments in the graph. It supports queries that fall into the following four categories (and their intersection): a) identification of individual arguments based on their content and structure, b) identification of structurally similar arguments, c) identification of different types of relations between arguments and d) identification of complete paths in the graph.

The results of ArgQL can be either individual values consisting of arguments and/or the components of arguments (i.e., premises or conclusions – called propositions), or more complex expressions that correspond to complete paths of arguments that match with the queries. Some examples or ArgQL queries follow:

- Description: Find arguments which have in their premises the proposition "Freedom means responsibility".

```
match ?a: <?p[/{"Freedom means responsibility"}], ?c>
return ?a
```

[2] http://discoursedb.org/

[3] http://www.arg-tech.org/wp-content/uploads/2011/09/aif-spec.pdf

[4] http://www.arg.dundee.ac.uk/aif

- Description: Find pairs of arguments for which, the premises of the first is a subset of the premises of the second.
  match ?a: <?pr1, ?c1> , ?b: <?pr2[/?pr1], ?c1>
  return ?a, ?b
- Description: Find and return the complete path of arguments (?a, ?c, ?b), such that ?a attacks ?c, ?c supports ?b and ?b has conclusion “Freedom means responsibility”.
  match ?a attack/support ?b: <?pr,"Freedom means responsibility">
  return path(?a, ?b)

The implementation of ArgQL is based on AIF and SPARQL. In particular, we assume that the argumentative information has been encoded in RDF format under the AIF specification. Then, each ArgQL query is translated into a SPARQL one that returns the triples that describe the answer to the original ArgQL query. Finally, these triples are translated into a more human-readable form (which uses argumentation terminology and is representation-agnostic) before being returned to the user.

3. Extending ArgQL with Keyword Search and Metadata

3.1. Keyword Search

Argument patterns constitute the fundamental elements that are used to match arguments in ArgQL. One of the ways to filter arguments is through string matching, but the original specification only allowed for exact string matching on propositions that appeared in argument patterns, and this was highlighted as one of the language’s shortcomings [9, 11]. In the proposed ArgQL extension, a more generic keyword search functionality can be used to filter arguments whose premise and/or conclusion contains a keyword, while supporting wildcards to allow non-exact matching.

To support keyword search at the syntactic level, we reused the existing argument pattern mechanism of ArgQL, which allows searching based on the text of the premise and/or conclusion of the argument. In the original specification, the argument pattern <?pr, “text”> would identify arguments with conclusion being exactly “text”; analogously, the argument pattern <?pr[/{“text”}], ?c> would identify arguments whose premise set contains the premise “text”.

We extend this idea and allow argument patterns of the above form to match triples in which the “text” string is contained within the conclusion/premise respectively, in a case-insensitive manner. We also allow the special character ‘*’ after the keyword, to denote that the conclusion/premise should contain text starting with the keyword.

For instance, a query to return all the arguments that contain any word starting with "Rich", "rich", "Richard", "richie" etc. in their conclusion would be:

```
match ?a: <?p, "rich*">
return ?a
```

To implement this new functionality, we reused the existing translation mechanism of ArgQL into SPARQL [10]. Table 1 shows how the ArgQL query presented in the above example is translated into its respective SPARQL.

Table 1
Keyword-search over conclusions

match ?a: <?p, “rich*”> return ?a	SELECT * WHERE { ?_i1 aif:claimText ?_prem_txt. ?_i1 aif:Premise ?_ra1. ?_i2 aif:claimText ?_conc_txt. filter(regex(?_conc_txt, "^rich/i")). ?_ra1 aif:Conclusion ?_i2. ?_ra1 rdf:type aif:RA-node. }

Note that the keyword search could be implemented with the use of the regex filter (as shown in Table 1), which is a generic SPARQL feature. However, in big datasets, such an implementation could

have performance issues. To address this, different triplestores contain optimized structures for keyword searching, which could be employed by the ArgQL implementation, if the underlying triplestore is known at design time.

In the context of DebateLab we used the Virtuoso Triplestore[5], exploiting its full-text index[6] to achieve a very good performance in the full text search using bif:contains, a specialized Virtuoso keyword that replaces the SPARQL's generic regex filter. As a result, the translated SPARQL query of Table 1 would actually be written as shown in Table 2 in the context of DebateLab. This also explains our syntactic choice (using "rich*" rather than "^rich/i") for ArgQL keyword search.

Table 2

Keyword-search over conclusions using Virtuoso triplestore

match ?a: <?p, "rich*"> return ?a	SELECT * WHERE { ?_i1 aif:claimText ?_prem_txt. ?_i1 aif:Premise ?_ra1. ?_i2 aif:claimText ?_conc_txt. filter(bif:contains(?_conc_txt, "rich*")). ?_ra1 aif:Conclusion ?_i2. ?_ra1 rdf:type aif:RA-node. }

3.2. Metadata

In the original ArgQL specification, the main focus for searching was the arguments themselves, or paths of arguments. However, arguments may be associated with attributes in the form of metadata (e.g., the date the argument was created, the author etc), which may be of interest to the user, either as an argument filtering mechanism, or to be returned as part of the query result. To support this, we introduce the notion of metadata that refer to arguments, and are essentially:

- Datatype properties referring to arguments, such as the author of the argument, the date of its creation etc.
- Paths of properties which lead to datatype properties such as the topics or the title of the document which contains the corresponding argument.

Querying metadata is a versatile tool, which can be used in different ways. In particular, any type of property or path of properties associated with an argument can be classified as "metadata", allowing ArgQL to consider it.

A metadata filter is essentially a pair of the form (metadata: expression). The type of metadata determines the allowed expressions to be used in the argument pattern:

- Metadata that refer to numeric and date constants support comparison operators (i.e., >, <, >=, <=, !=, =), as well as operators which define a range of values either exclusively (i.e., (…)) or inclusively (i.e., […]).
- Metadata that refer to string constants support keyword-based search.

Finally, we can have combinations of filters with conjunctions (&&) or disjunctions (||). Next, we provide some examples to show how ArgQL is extended to accept metadata filters. For our examples we will use two metadata properties, namely, the creationDate (tm) and the argTitle (tit), which denote the date the argument was created and the title of the document where the argument is contained respectively:

- Find arguments with a creation date in April 2022
  match ?a: <?pr, ?c> [tm: "[2022-04-01, 2022-04-30]"]
  return ?a
- Find arguments within articles whose title contains the keyword "airport"

[5] https://virtuoso.openlinksw.com

[6] http://docs.openlinksw.com/virtuoso/rdfsparqlrulefulltext

```
match ?a: <?pr, ?c> [tit: "airport"]
return ?a
```

- Find arguments which were created after 2022-04-01 and are contained in an article whose title contains the keyword “airport”

```
match ?a: <?pr, ?c> [tm: ">=2022-04-01" && tit: "airport"]
return ?a
```

As already mentioned, metadata are not only useful as a filtering tool, but can also be returned along with the arguments’ information. To support this functionality, we extend the form of the “return” block of ArgQL as follows:

```
return ?a, metadata_name_1(?a), metadata_name_2(?a), …
```

As an example, if we want to return all arguments in the knowledge base, along with their creation date (tm) and title of containing document (tit), we should write:

```
match ?a: <?pr ,?c>
return ?a, tm(?a), tit(?a)
```

Finally, apart from returning the metadata, we can also sort the results with respect to one or more metadata, in ascending or descending order, by using an order-by expression. If we omit the order type, we get ascending order by default. The form of the order-by expression is:

```
order by metadata_name_1(?a) {ASC/DESC}, metadata_name_2(?a) {ASC/DESC}, …
```

In the above example, if we wanted arguments to be ordered with respect to their creation date in a descending order, we would write:

```
match ?a: <?pr ,?c>
return ?a, tm(?a), tit(?a)
order by tm(?a) DESC
```

As mentioned, metadata are essentially datatype properties or paths of properties that lead to datatype values. Thus, for the translation of the ArgQL query into the corresponding SPARQL, we just have to include the corresponding triple patterns in the SPARQL containing the required metadata filter. Since we support any type of metadata, these triple patterns are not known at design time and should be provided at initialization time (through a configuration file) to the ArgQL implementation. This configuration file essentially maps each metadata type to a metadata definition that is a set of triple patterns which should be included into the translated SPARQL query to identify the respective metadata. Table 3 shows the translated SPARQL query in the case of a metadata date filter in which we require the creation date of the argument to be after 2022-04-01.

Table 3

Extended ArgQL with date filters

match ?a: <?pr, ?c>
[tm: ">=2022-04-01"]
return ?a
	SELECT * WHERE {
	?_i1 aif:claimText ?_pr_txt.
	?_i1 aif:Premise ?_ra1.
	?_i2 aif:claimText ?_conc_txt.
	?_ra1 aif:Conclusion ?_i2.
	?_ra1 rdf:type aif:RA-node.
	?_ra1 aif:creationDate ?_ra1_tm.
	filter (xsd:datetime(?_ra1_tm) >= xsd:datetime("2022-04-01")).
	}

For the returning of metadata, no extra treatment is required with regards to the SPARQL query, as they are already returned (due to SELECT *, see Table 3). If we want to order the resulting arguments with respect to a metadata type, we should add the “ORDER BY” expression in the respective SPARQL as shown in Table 4.

Table 4
Extended ArgQL with metadata filters and metadata values returned sorted

```
match ?a: <?pr, ?c>          SELECT * WHERE {
[tm: ">=2022-04-01"]         ?_i1 aif:claimText ?_pr_txt.
return ?a, tm(?a)            ?_i1 aif:Premise ?_ra1.
order by tm(?a) asc          ?_i2 aif:claimText ?_conc_txt.
                             ?_ra1 aif:Conclusion ?_i2.
                             ?_ra1 rdf:type aif:RA-node.
                             ?_ra1 aif:creationDate ?_ra1_tm.
                             filter ( xsd:datetime(?_ra1_tm) >= xsd:datetime("2022-04-01") ).
                             } ORDER BY ASC(?_ra1_tm)
```

Note that, in order for this functionality to work, the metadata values need to be stored in the underlying Knowledge Graph. Since we adopt an open architecture, allowing any type of developer-defined metadata to be supported, AIF does not necessarily provide features to store this information, and appropriate additional properties need to be defined. A configuration file is used to associate such properties with the respective metadata, allowing ArgQL to be extended with any arbitrary metadata that are needed for the application at hand. In the context of DebateLab, such metadata are included in the DebateLab database at ingestion time, thereby allowing the use of ArgQL to query this information.

4. Extended ArgQL Syntax

We briefly mentioned above the syntactic extensions of ArgQL to support keyword searching and metadata querying. Here, we provide a more complete description, in the form of a BNF grammar (see Table 5), clearly showing (in italics and underlined font) the additions to the original BNF provided in [9].

More specifically, for the keyword search, we extended the proposition expression to consider the starts-with keyword search (if the character '*' is present) as we are using Virtuoso's bif:contains property.

For the metadata, we had to introduce some new expressions in order to be able to recognize the metadata definitions. First, we had to extend the argpattern expression by adding the metadata expression namely, md_express. A metadata expression consists of a set of metadata filters (md_filter) combined with conjunctions (&&) or disjunctions (||). Finally, a metadata filter consists of a metadata variable name (md_name) and a filter, which, as mentioned, depends on the type of the metadata (see expressions num_filter, rang_filter, keyw_filter). Finally, considering that the metadata values can also be returned along with the arguments' information, we extended the returnvalue expression with a set of metadata (md_return_val) with an optional ascending or descending order.

Table 5
Extended ArgQL syntax (reserved words in bold, new extensions in underlined italics)

```
                query ::= 'MATCH' (dialoguepattern (',' dialoguepattern )*
                          'RETURN' returnvalue (',' returnvalue)*
  dialoguepattern ::= argpattern |
                          argpattern pathpattern dialogue_pattern
        argpattern ::= variable |
                          (variable:)?'<'premisepattern','
                                  concluspattern'>' md_express?
     premisefilter ::= '[' ('/' | '.' ) (propset | variable) ']'
   concluspattern ::= variable | proposition
          propset ::= '{' proposition (',' proposition)* '}'
      pathpattern ::= pp ('/' pp )*
               pp ::= relation |
```

	'**(**' pathpattern '**)**' ('***' | '**+**') num
returnvalue ::=	variable | '**PATH**' '**(**' variable ',' variable ')' |
	*variable (',' md_val)**
	*('**ORDER BY**' md_val_ord (',' md_val_ord)*)?*
relation ::=	**'attack'** | **'rebut'** | **'undercut'** |
	'support' | **'endorse'** | **'back'**
proposition ::=	variable | string | *string ('*')?*
variable ::=	'?'('a'…'z' | 'A'…'Z' | '0'…'9')+
string ::=	'""'.*?'""'
md_express ::=	***'['** md_filter ('**&&**'|'**||**' md_filter)* **']'***
md_filter ::=	*md_name ':' (num_filter|rang_filter|keyw_filter)*
md_name ::=	*(a-zA-Z)+ (a-zA-Z0-9_)**
md_num_op ::=	*('**>**'|'**<**'|'**>=**'|'**<=**'|'**=**'|'**!=**')?*
num_filter ::=	*'"' md_num_op number '"'*
rang_filter ::=	*'"'('(' | '[') number ',' number (')' | ']')'"'*
keyw_filter ::=	*string*
number ::=	*(('0'…'9')'.')? ('0'…'9')+*
md_val ::=	*md_name'(' variable ')'*
md_val_ord ::=	*md_val (**'ASC'**|**'DESC'**)?*

5. Conclusion and Future Work

We presented two extensions of the ArgQL specification, namely the keyword search functionality and the metadata querying functionality. Both extensions constitute significant components of most query languages (especially in the context of the Semantic Web), but were lacking from the original ArgQL specification. Thus, we argue that they enhance ArgQL's expressive power in meaningful ways, and believe they will assist users in addressing more complex and intuitive information needs.

As a future step, we plan to extend the new functionalities to consider content equivalences (rephrasings), i.e., cases where two different propositions express the same thing in different words, a common scenario in real-world argumentation. We also plan to provide an efficient implementation over a specific Triplestore. As DebateLab is dealing with the domain of e-journalism, we are interested in using real life data from that domain and conduct experiments to see both the performance of our implementation and the usefulness of the provided results for our end users (i.e., journalists). Additional useful features could include the implementation of a tool to support naïve and/or advanced users to write their own ArgQL queries.

Acknowledgements. This work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "1st Call for H.F.R.I. Research Projects to support Faculty Members and Researchers and the procurement of high-cost research equipment" (Project #4195).

6. References

[1] C. Chesnevar, J. McGinnis, S. Modgil, I. Rahwan, C. Reed, G. Simari, M. South, G. Vreeswijk, and S. Willmott. Towards an argument interchange format. Knowledge Engineering Review, 21(4):293–316, 2006.

[2] P. M. Dung. On the Acceptability of Arguments and Its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-person Games. Artificial Intelligence, 77(2), 1995.

[3] J. Lawrence, C. Reed. AIFdb Corpora. In: S. Parsons, N. Oren, C. Reed, F. Cerutti (eds.) Computational Models of Argument, pages 465-466, 2014.

[4] I. Rahwan and G. Simari. Argumentation in Artificial Intelligence. Springer, 2009.

[5] I. Rahwan, F. Zablith and C. Reed. Laying the Foundations for a World Wide Argument Web. Artificial Intelligence, 171(10-15):897-921, 2007.

[6] C. Reed, S. Wells, J. Devereux and G. Rowe. AIF+: Dialogue in the Argument Interchange Format. Frontiers in artificial intelligence and applications, 172, p.311, 2008.
[7] J. Schneider, T. Groza, and A. Passant. A Review of Argumentation for the Social Semantic Web. Semantic Web Journal, 4(2):159-218, 2013.
[8] E. Ymeralli, G. Flouris, T. Patkos, D. Plexousakis. APOPSIS: A Web-based Platform for the Analysis of Structured Dialogues. In Proceedings of the 16th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE-17), 2017
[9] D. Zografistou. ArgQL: Querying Argumentative Dialogues using a Formal, Structured Language (PhD Thesis), Computer Science Department, University of Crete, 2019.
[10] D. Zografistou, G. Flouris, T. Patkos, D. Plexousakis: Implementing the ArgQL Query Language. COMMA 2018: 241-248.
[11] D. Zografistou, G. Flouris, D. Plexousakis. ArgQL: A Declarative Language for Querying Argumentative Dialogues. In Proceedings of the International Joint Conference on Rules and Reasoning 2017 (RuleML+RR), as the Best Paper of the Doctoral Consortium, 2017.