Deepflow: Using Argument Schemes to Query Relational Databases

Jann MÜLLER [a,1] and Anthony HUNTER [b]
[a] *SAP UK, Belfast BT3 9DT, United Kingdom, jann.mueller@sap.com*
[b] *University College London, London WC1E 6BT, a.hunter@cs.ucl.ac.uk*

Keywords. argument schemes, argumentation, domain-specific language, SQL

Project DEEPFLOW aims to extract and analyse the rationale behind design decisions from textual documentation. Once the data has been extracted and stored in a relational database, it can be queried with the "dashboard" component described here. The dashboard provides a language for writing argument schemes [3] and critical questions as a means to query data about a series of decisions related to a project (what was decided, who decided it, what were the reasons, which alternatives were considered etc).

Users can create argument schemes and critical questions based on a set of primitive types determined by the entities and relations in the underlying database. Argument schemes are thus not part of the database schema, but instead they are defined dynamically and on-line. They are used not to structure the database, but to query it. One can experiment with varying definitions of argument schemes to answer questions such as

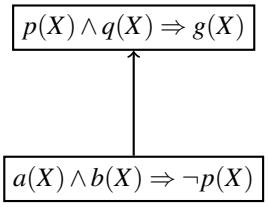
1. "How many decisions were supported by expert advice in 2012?"
2. "Which claims were rejected because they were made by experts with fewer than 5 publications?"
3. "How does that number change if I lessen the restriction to 3 publications?"

The user interface shows a code editor where users can see and edit the list of argument schemes and critical questions, similar to a Prolog program. Some aggregations are available, such as `COUNT` and `MAX`, which can be used in the definitions of argument schemes. This allows the user to explore the data base by experimenting with different argument schemes. For example, the argument scheme "Expert Opinion" (simplified) is written as

$$\mathsf{expert}(\mathsf{X}) \wedge \mathsf{claim}(\mathsf{Y}) \wedge \mathsf{claims}(\mathsf{X},\mathsf{Y}) \Rightarrow \mathsf{accept}(\mathsf{Y}) \tag{1}$$

Critical questions are rules whose claim negates one of the premises of an argument scheme. For example to specify that experts are only those with at least five recorded publications one can write $\mathsf{COUNT}(\mathsf{publication}(\mathsf{X})) < 5 \Rightarrow \neg\mathsf{expert}(\mathsf{X})$. It is also possible to define exceptions to critical questions, so the depth of the resulting dialectical tree is not limited.

[1] This work is supported by SAP AG and the Invest NI Collaborative Grant for R&D - RD1208002.



Step 1 Negate the condition of the critical question

$$S_1 = \{\neg(a(X) \wedge b(X)) \wedge p(X) \wedge q(X) \Rightarrow g(X)\}$$

Step 2 Simplify and remove disjunctions

$$S_2 = \{\neg a(X) \wedge p(X) \wedge q(X) \Rightarrow g(X),$$
$$\neg b(X) \wedge p(X) \wedge q(X) \Rightarrow g(X)\}$$

Figure 1. Transforming the dialectical tree of an argument scheme with a critical question (left) to a set of rules S_i.

By translating argument schemes to SQL the task of analysing and selecting records is offloaded to the database, which is optimised for exactly such queries. The client is only concerned with creating queries that represent argument schemes.

The results are visualised in several graphics. A second way of querying the data (besides writing argument schemes) is to add additional filters by "drilling down", ie clicking on one of the data points in the visualisations. This allows for a hybrid exploration of the data, using both argument schemes and non-argumentative attributes.

The process of translating argument schemes with critical questions to SQL queries consists of three steps. Steps 1 and 2 (see Fig. 1) turn dialectical trees into sets of "flat" rules and are repeated until the entire tree has been consumed. In Step 3 each of the flat rules is interpreted as a SQL query. To illustrate the last step, the SQL query generated for the scheme (1) is given below (critical questions omitted for brevity).

```
SELECT * FROM claim as Y
JOIN makesClaim as C on C.claimId = Y.claimId
JOIN expert as X on X.expertId = C.expertId
```

The conclusion of (1), accept(X), does not correspond to a table in the database. This shows how the language can be used to create new predicates from primitives. Non-primitive predicates such as accept are modeled as virtual tables (views) on the database, to enable re-use of intermediate results. Negated predicates are interpreted as EXCEPT clauses in SQL, making use of the closed world assumption for relational data.

The DEEPFLOW dashboard is written in Haskell, which allows us to leverage an existing implementation [2] of abstract argument graphs [1]. The server in our case is an SAP HANA database, but the techniques described here are not platform-specific and work with any SQL database.

References

[1] Dung, P.: On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artificial Intelligence* **77** (1995), 321–357

[2] van Gijzel, B.: Tools for the implementation of argumentation models. *ICCSW, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany* **35** (2013), 43–48

[3] Walton, D.; Reed, C. & Macagno, F.: *Argumentation Schemes*, Cambridge University Press, 1994