UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

ArgQL: Querying Argumentative Dialogues using a Formal, Structured Language

by

Dimitra Zografistou

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, November 2019

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

ArgQL: Querying Argumentative Dialogues using a Formal, Structured Language

PhD Dissertation Presented

by **Dimitra Zografistou**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

[illegible]

Author: Dimitra Zografistou

[illegible]

Supervisor: Dimitris Plexousakis, Professor, University of Crete

[illegible]

Committee Member: Giorgos Flouris, Principal Researcher, FORTH-ICS

[illegible]

Committee Member: Antonis Bikakis, Associate Professor, University College of London (UCL)

[illegible]

Committee Member: Grigoris Antoniou, Professor, University of Huddersfield

[illegible]

Committee Member: Chris Reed, Professor, University of Dundee

[illegible]

Committee Member: Anthony Hunter, Professor, University College of London (UCL)

[illegible]

Committee Member: Fouad Zablith, Assistant Professor, American University of Beirut

[illegible]

Department Chairman: Bilas Angelos, Professor, University of Crete

Heraklion, November 2019

To my family...

Acknowledgments

The years of my PhD studies have been recorded in my memory as a blending of both good and difficult moments, related to the work itself or to personal concerns and motivations. In all those moments, I had in my surroundings people, who consciously or unconsciously and each one in his own way offered valuable support. There are no right words to express how lucky I feel for that. This page is particularly devoted to them.

First and foremost, I would like to express my sincere gratitude to my supervisor *Prof. Dimitris Plexousakis* for giving me the opportunity to attend the PhD program, for his insightful advises and for being that supportive whenever someone needs him. I owe a lot not only to his scientific experience, but mainly to the quality of his character.

I would also like to thank the rest of my examination committee, *Dr. Antonis Bikakis*, also one of the three members of my Advisory Committee, who followed this work from the first steps and gave valuable comments and remarks, as well as *Prof. Anthony Hunter*, *Prof. Grigoris Antoniou* and *Dr. Fuad Zablith* for taking the time to participate in the examination and for the interesting comments and questions. At this point, I would particularly like to refer to *Prof. Chris Reed* for doing all this trip in order to join the examination physically, for the interest he manifested about my work and for the opportunity he gave me to continue my research activity in collaboration with his team.

This thesis wouldn't have been completed in the way it did without the contribution of *Dr. Giorgos Flouris*, also member of my Advisory Committee. Giorgos combines enthusiasm and intelligence and working with him was in many aspects a gainful experience. I am deeply thankful for all of his support, for getting too much into the problems of this work and most importantly for believing in it – sometimes even more than I did.

Deep gratitude goes to *Dr. Theodore Patkos* for the endless and stimulating discussions (both the scientific and the more personal ones). Theodore has a unique way to motivate someone to go after the best for him and I feel the need to thank him for his precious advises.

I would like to acknowledge the Institute of Computer Science at FORTH and the Computer Science Department of the University of Crete for the excellent working environment and the financial support during the period of my studies.

I can't tell how much I thank my beloved friend (and office neighbor), *Roula Avgoustaki*. Her vivid personality, great sense of humor and warm character made the days at work feel so funny and enjoyable that turned out to be an important support against any difficulty.

During the years of my presence in the Information Systems Lab of ICS-FORTH, I had the chance to meet exceptional colleagues with which we shared our everyday life, and some of them also became friends. I genuinely thank them all for offering such a great environment to work and especially

Katerina Papantoniou for the discussions and her support, *Constantinos Varsos* for all the fun that we had, *Panagiotis Papadakos* for his advises, always expressed from his own distinctive point of view, *Fillipos Gouidis* for the literary moments he offered, the (now and ex) neighbors, *Kostas Petrakis* and *Johny Chrysakis* for sharing great moments in the hub, *Georgia Troullinou, Konstantina Konsolaki, Nicole Tsampanaki, Nina Saveta, Giannis Roussakis, George Samaritakis* for the unforgettable lunch and coffee breaks, as well as *Maria Moutsaki* for making the life with the paperwork so much easier and *Dimitris Angelakis* for all the technical support.

Beyond the physical boundaries of FORTH, there have been people, whose presence in my life has proven to be invaluable. In particular, I refer to: *Nikolas Grigoriou* for being a brotherly friend since we were kids, *Giorgos Kalomoirakis* for all the things he has done for me and for being the person he is, *Ioanna Zerva* for her way of thinking, which helped me so many times and for giving me the delight to become godmother of an adorable child, *Athina*, *Eirini Genitsaridi* for our great discussions and her clear point of view and *Larissa Sidorova* for her jolly and supportive character that helped me get away from anything occasionally concerned me. Also to friends, with which I have spent precious moments in the past and, although apart, are always in my heart. Especially *Mary Koutraki* for our longstanding friendship, *Sofia Kleisarchaki* for her open-minded approach to the situations, *Evangelia Mavromihelaki* for her willingness to help anyone whenever needs it, *Nelly Vouzoukidou* for her always positive attitude and *Maria Valentaki*, for the friendship which is never affected by any space or time distance. Finally, I am truly thankful to *Nikos Mastrogiorgakis* for all those beautiful features of his character, for believing in me, his encouragement and support and, even if it is not a long time, I already feel really lucky to have him in my life. Words are not enough to express my gratitude. I love you all, each one in a different way and for different reasons.

Last in this page but always first in my heart is my family. I am deeply and forever grateful to my parents *Antonis* and *Eleni* for raising me to the person I am today, for their endless love and support and for being constantly but subtly next to me in all of my decisions. Also my sisters, *Chrysoula* and *Eva*, my beloved grandma, *Pelagia* and the wonderful nieces *Chrysoula* and *Spiros* gifted me, *Ariadne* and *Nepheli* (and the youngest one which is on the way :)), for being the persons I can always count on and I will always be there for them. I wouldn't have made it without you and therefore this thesis is dedicated to you..

Abstract

In light of the rapid evolution of social media that we experience over the past decade, and their establishment as one of the main means of communication and dialogical exchange, the problem of extracting meaningful information from data residing in human dialogues is now more crucial than ever. Until recently, the challenges associated with this problem were being addressed as an application domain for the computational models of fields like the Semantic Web, Information Retrieval, Data Mining etc. However, there are some information requirements when searching in dialogues which are quite specific and common for all types of dialogues, regardless of their context or goal, e.g. concerning the structure of the different opinions and the correlations among them, that could stand as an autonomous area to study. Isolating those requirements and bringing them together in the specification of a formal language, designed exclusively for this purpose, is a research direction which has been given less attention.

Incited by this deficiency, in this thesis we introduce *ArgQL (Argumentation Query Language)*, a high-level declarative language for querying dialogical data. ArgQL provides a simple and dialogue-related terminology to write queries in the domain, which in existing query languages would be quite difficult to express. The theory that founds the data model adopts some of the most prevailing semantics in the area of Computational Argumentation. As a result, the target data consist of graphs of interconnected, structured arguments and ArgQL allows for the navigation across such graphs. We present the formal specification of the language, including the definition of its main constructs, the concrete syntax, as well as the semantics that determine the evaluation of those constructs against the data model.

Subsequently, we propose a methodology to translate ArgQL into other languages and in particular we show the case of RDF and its associated query language, SPARQL. To this end, we define an RDF scheme based on the AIF conceptualization and we formalize the mappings between this and our data model. We then build the process of query translation, as a set of rules, that define the correspondence between the different ArgQL constructs and the respective SPARQL graph patterns. The soundness and completeness of the process is verified by proving the equivalence between the matching data for each of these two languages, with respect to their formal semantics. Although correct, the conformation to the precise definition of the translation rules results to non-optimal queries. Therefore, on top of this methodology, we propose some optimization, that succeeds shorter and by extension more efficient queries. We also give prominence to the practical side of ArgQL and we implement the language, allowing for its execution on real data-sets. Thus, one of the outcomes of this work was an online query endpoint, where someone can test his own queries. Finally, we conduct an experimental study to evaluate the query performance under different execution parameters.

Keywords: Computational Argumentation, Graph Query Languages, AIF, Semantic Web

Supervisor:
Dimitris Plexousakis
Professor of Computer Science Department
University of Crete

Περίληψη

Μπροστά στην ραγδαία εξέλιξη των κοινωνικών δικτύων που βιώνουμε την τελευταία δεκαετία και της αδιαμφισβήτητης επικράτησής τους ανάμεσα στα διαφορετικά μέσα επικοινωνίας, το πρόβλημα της ανάκτησης ωφέλιμης πληροφορίας από διαλογικά δεδομένα είναι τώρα πιο κρίσιμο από ποτέ. Μέχρι πρόσφατα, οι προκλήσεις που συνοδεύονταν με το συγκεκριμένο πρόβλημα αντιμετωπίζονταν ως ένα επιπλέον πεδίο εφαρμογής των υπολογιστικών μοντέλων που έχουν αναπτυχθεί σε τομείς όπως αυτόν του Σημασιολογικού Ιστού, της Ανάκτησης Πληροφορίας της Εξόρυξης Δεδομένων κλπ. Παρόλα αυτά, υπάρχουν κάποιες πληροφοριακές ανάγκες που σχετίζονται με το πρόβλημα της αναζήτησης πληροφορίας σε διαλόγους, οι οποίες είναι κοινές για όλους τους τύπους διαλόγων, ανεξαρτήτως του ευρύτερου πλαισίου τους ή του στόχου που μπορεί να έχουν, πχ. που αφορούν τη δομή των διαφορετικών απόψεων και των πιθανών συσχετίσεων μεταξύ τους, οι οποίες θα μπορούσαν να σταθούν σαν μία ανεξάρτητη περιοχή προς μελέτη. Η απομόνωση αυτών των αναγκών και η ένταξή τους στον προσδιορισμό μίας τυπικής γλώσσας, η οποία θα έχει σχεδιαστεί αποκλειστικά γι αυτό το σκοπό, αποτελεί μία ερευνητική κατεύθυνση, η οποία έχει μέχρι τώρα μελετηθεί ελάχιστα.

Παρακινούμενοι από αυτή την έλλειψη, σε αυτή την εργασία παρουσιάζουμε την ΓΕΕ (Γλώσσα Επερωτήσεων Επιχειρηματολογίας), μία υψηλού επιπέδου, δηλωτική γλώσσα για την υποβολή ερωτημάτων σε διαλογικά δεδομένα. Η ΓΕΕ προσφέρει μία απλή και σχετιζόμενη με διαλόγους ορολογία για την σύνταξη ερωτημάτων στο συγκεκριμένο τομέα, τα οποία, με τις υπάρχουσες γλώσσες επερωτήσεων, θα ήταν αρκετά δύσκολο να εκφραστούν. Η θεωρία που θεμελιώνει το μοντέλο δεδομένων υιοθετεί κάποια από τα επικρατέστερα σημασιολογικά μοντέλα, που έχουν οριστεί στην περιοχή της Υπολογιστικής Επιχειρηματολογίας. Συνεπώς, τα δεδομένα στα οποία απευθύνεται η γλώσσα αποτελούνται από γράφους διασυνδεδεμένων και δομημένων επιχειρημάτων και η ΓΕΕ δίνει τη δυνατότητα της πλοήγησης σε τέτοιους γράφους. Παρουσιάζουμε λοιπόν τις τυπικές προδιαγραφές τις γλώσσας, οι οποίες περιλαμβάνουν τον ορισμό των βασικών της δομών, το συντακτικό της καθώς επίσης και το σημασιολογικό μοντέλο, βάσει του οποίου καθορίζεται η αποτίμηση των δομών αυτών με στοιχεία του μοντέλου δεδομένων.

Ακολούθως, προτείνουμε μία μεθοδολογία για τη μετάφραση της ΓΕΕ σε άλλες γλώσσες, και συγκεκριμένα, δείχνουμε την περίπτωση της ΡΔΦ (Γλώσσα Περιγραφής Πηγών) και της συσχετιζόμενης με αυτήν γλώσσας επερωτήσεων, ΣΠΑΡΚΛ. Προς αυτή την κατεύθυνση, ορίζουμε ενα ΡΔΦ σχήμα το οποίο βασίζεται στο εννοιολογικό μοντέλο της οντολογίας ΑΙΦ και παρουσιάζουμε την αντιστοίχιση μεταξύ αυτής της αναπαράστασης και του δικού μας μοντέλου δεδομένων. Στη συνέχεια, παρουσιάζουμε τη διαδικασία της μετάφρασης των επερωτήσεων, ως ένα σύνολο κανόνων που ορίζουν την αντιστοιχία μεταξύ των διαφορετικών δομών της ΓΕΕ και των αντίστοιχων μοτίβων γράφου στην ΣΠΑΡΚΛ. Οι ιδιότητες της ορθότητας και πληρότητας της διαδικασίας, επαληθεύονται αποδεικνύοντας την ισοδυναμία μεταξύ των αποτελεσμάτων για καθεμία από τις δύο γλώσσες, αναφορικά και με την τυπική σημασιολογίας τους. Παρά την ορθότητα των κανόνων μετάφρασης, η ακριβής εφαρμογή τους οδηγεί σε μη βέλτιστες επερωτήσεις. Γι αυτό το λόγο, πάνω σε αυτή τη μεθοδολογία, προτείνουμε κάποιες βελτιστοποιήσεις, οι οποίες επιτυγχάνουν μικρότερα σε μήκος και κατ΄ επέκταση πιο αποδοτικά ερωτήματα. Αναδεικνύουμε ακόμα την πρακτική πλευρά της ΓΕΕ υλοποιώντας τη γλώσσα και δίνοντας της τη δυνατότητα να εκτελείται σε πραγματικές συλλογές δεδομένων. Έτσι, ένα από τα αποτελέσματα αυτής της δουλειάς, είναι μία εφαρμογή του ιστού, στην οποία μπορεί οποιοσδήποτε να δοκιμάσει να τρέξει τα δικά του ερωτήματα. Τέλος, διεξάγουμε μία πειραματική μελέτη, προκειμένου να αξιολογήσουμε την απόδοση των επερωτήσεων κάτω από διαφορετικές παραμέτρους εκτέλεσης.

Λέξεις κλειδιά: Υπολογιστική Επιχειρηματολογία, Γλώσσες Επερωτήσεων σε Γράφους, ΑΙΦ οντολογία, Σημασιολογικός Ιστός

Επόπτης:
Δημήτρης Πλεξουσάκης
Καθηγητής Τμήματος Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Contents

1.1 The big picture

Deliberation has always been the means for humans to manifest their inherent need for communication and self expression. Everyday, people engage in discussions stating their opinions about mulitple issues like their daily routine, current affairs, sociopolitical problems, scientific progression and uncountable other areas. A dialogue is usually incited by specific goals like the resolution of a conflict, searching for a mutual settlement, exchanging information, reaching common decisions etc. An interesting categorization in the different types of a dialogue can be found in [119]. The final goal may or may not be common for all of the participants and this usually depends on their individual motives and aspirations, factors that also affect implicitly the course of a discussion.

The advancements in the technologies of Web 2.0 over the past decade unleashed new potentials to the way humans communicate. Among others, Web turned into a communication venue of a universal expansion, changing the way in which users interact with it and transforming them from the passive content consumers they were during the Web 1.0 era, into active content generators. Nowadays, users are given the opportunity to join online communities, get involved in public discussions with other users around the world, express in public their opinion about particular issues and give feedback on published information. Social networks (e.g. Facebook[1]), blogs and microblogs (e.g.

[1] www.facebook.com

Twitter[2]), debate portals and crowd-sourcing platforms(e.g. Debate.org[3] , CreateDebate[4]) and websites allowing rating and reviewing products (e.g. Amazon[5]) are only a few examples enabling online discussions. Figure 1.1 gives an insight about the generation rate of user content by some of the most popular social platforms.

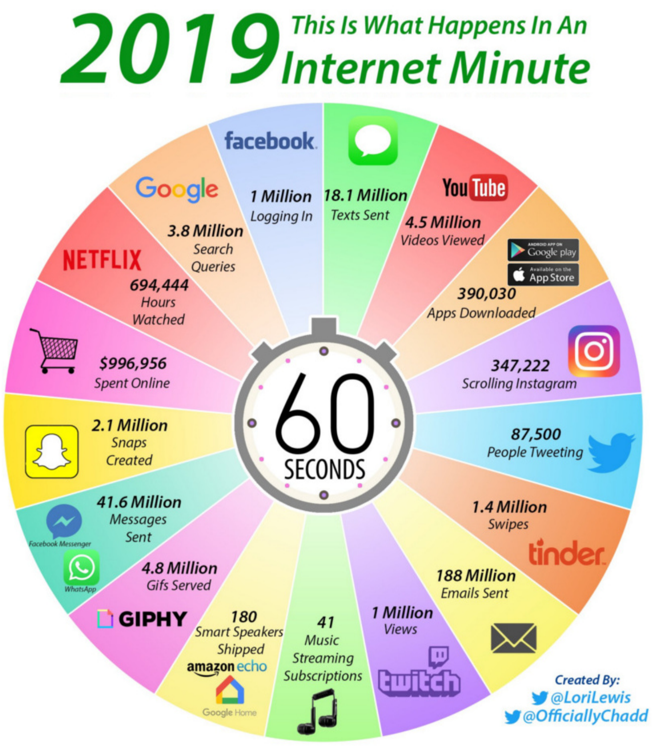


Figure 1.1: Infographic: What happens in an Internet minute in 2019?

1.2 Motivation and Problem Description

As shown in the figure 1.1, tons of data are published every minute on the web. The digital format of this data varies from pure text, to audiovisual, sharing though one common feature: the lack of structure and semantic characterization, an absence which harms their potential for utilization. A considerable portion of these volumes is owned to activities like commentary and discussions between users. Things become more complicated with the interpretation of such data and this is due to the vagueness and subjectivity that characterize their nature which make it hard even for a human expert to analyze and draw conclusions about them. However, the abundance of information lying within

[2]www.twitter.com
[3]https://www.debate.org/
[4]http://www.createdebate.com/
[5]www.amazon.com

this data, and the fact that it could offer valuable material to many different disciplines and fields like journalism, statistics, law etc. is something that should not be overlooked, making the need to find ways to structure, analyze and retrieve them, more imperative than ever.

Under the veil of this ambition, we deal with the problem of sense-making and extracting meaningful information over human-dialogical data. The ascertain that the information requirements associated with this problem are quite specific, discrete and also common for all dialogues, regardless of their context or goal, motivated us to isolate and investigate them as an independent domain. We discuss about requirements like exploration of the different ways opinions are justified or withdrawn, the identification of interactions and degree of relevance between different expressed positions, navigation across a dialogue, finding the most prevailing points in a discussion, evaluating the validity of positions or whether the goal of the dialogue has been reached also taking into account the personal motives of the participants etc.

Until now, the problem of knowledge extraction from human dialogues has been an application domain for the the models of various research fields. Database Management Systems [93] and Semantic Web [17] have provided mechanisms to represent and query such data using a particular scheme that includes concepts related to the domain [66, 94, 117]. The Natural Language Processing (NLP) [35, 70, 71] a subfield of linguistics and computer science, the studies of which focus on processing textual data, have provided methods to extract argumentative structures from data expressed in natural language [30, 68, 69, 107], a research area called argument mining. At the same time, the process of human reasoning while arguing has been an object of longstanding theoretical studies, having also featured significant computational models in the area of Argumentation [91].

The literature missed an approach which will formalize the process of information seeking in structured dialogues, providing certain criteria designed exclusively for the domain, that would bring together the different information requirements into its formal specification. Such a mechanism would lie on top of any specific database which is currently used to store dialogical data, and will offer the ability to write formal expressions to require for particular piece of data from dialogues, using a pure syntax, with built-in dialogue-related keywords and terminology, that a non-expert in computer science will be able understand and compose by himself. The prominence of such a mechanism is amplified by the expressive complexity of traditional languages (e.g. SPARQL) to express even simple statements like ”How an argument that supports a particular position is withdrawn”. We discuss such things in detail in section 4.1.

To address this limitation, we introduce *Argumentation Query Language (ArgQL)* [124–126], a high-level language for querying structured dialogical data. ArgQL reflects our first steps in the endeavor to understand and embody information requirements like the ones described above, into the specification of a formal language. ArgQL goes beyond the state of the art, since, to the best of our knowledge, it is the first query language designed for this purpose.

1.2.1 Use case scenario

We give a potential scenario that highlights the usability of ArgQL. Assume a poll company, which intents to conduct a survey about an issue e.g. the living conditions for a typical household of a country. To this end, the company picks a representative sample of the population to participate in the research. Apart from the standard questionnaires that contain some predefined choices that they have to fill in, it also raises a discussion poll with a few questions and asks them to discuss the specific issues. The objective for this is that they will be able to obtain a spherical awareness about the existing social problems from the citizens' standpoint, identify the points in which the majority agrees, the most controversial issues and generally come into a deeper comprehension, as well as a clearer explanation about the results they gathered by the questionnaires.

Subsequently, the company asks from a few analysts to study both questionnaires and discussions and extract a set of summary reports that will aggregate the results from the survey. The collected discussions may consist of hundreds of expressed opinions and it would be a very time-consuming task for someone to read them all. Instead, what they do is gather the collected data, process them with appropriate Natural Language Processing (NLP) algorithms which will transform them from their initial textual, unstructured form, into structures allowing for their interpretation. Then, using ArgQL, they write queries that express their requirements in order to locate the desired information within the collected data. Some examples of the requests that would help them create their reports are the following: ”How do people justify the position that .. ?”, ”How many of them disagree with this position and for what reasons?”, ”Find positions which are relevant to a particular argument”, ”Which are the most persuasive (acceptable) positions from those expressed and why?”, ”Return complete lines of discussion that conform to a particular motif”. Thereby, they minimize the effort they have to put for each different task of their report, since the results they get from ArgQL are more targeted and structured.

1.3 Research Questions

In order to come up against the challenges of defining a new query language from scratch, we dealt with a number of sub-problems to which it is decomposed, which have both research and practical character. In particular, we had to make decisions about the following research (and technical) questions:

- How can arguments and dialogues be modeled in a representation appropriate for searching and retrieving them, while at the same time being general enough to support different types of arguments and debates? What kind of interactions can be determined between arguments ?

- Definition of the set information requirements supported by the language. In which general categories can they be classified?

- Decision about the main characteristics of the language? Would it be procedural like a programming language, declarative like query languages, rule-based or other? Depending on the particular decision, decide about the syntax (keywords, expressions, returning elements etc.)? Build it upon some already existing language and if yes, on which and for what reasons?

- Having defined the formal model of data as well as the concrete syntax of the language, how can the semantics of the language be defined so that they will show formally the way that the different expressions of the language are evaluated against the data model. Would we be based on SPARQL semantics, on First order logic interpretation function or some hybrid approach?

- Regarding the implementation of the language decide between building it from scratch or translating to another language:
  - In the first option, how can we build a pure storage scheme based on the suggested data model and how will we achieve an efficient implementation?
  - With the second option, we had to decide to which language ours would be translated. We would also have to show the mappings between our data model and the respective data scheme. How can we define formally the translation, so that we will be able to prove that it is sound and complete?

- Based on the choice about the language implementation, we also have to decide about how will we evaluate its efficiency. Where can we find data-sets, what to evaluate and how will we conduct the experiments?

1.4 The Approach

The definition of the formal specification of ArgQL was built according to the formal semantics of a data model for representing arguments and dialogues. In order to define this data model, we had in our disposal a multitude of theoretical models coming from the area of Computational Argumentation [91]. Argumentation is a rich interdisciplinary area, ramified in various fields like Artificial Intelligence, collaborative learning, philosophy, linguistics and psychology. For at least two decades, its studies have served a wide variety of theoretical and computational reasoning models that simulate human cognitive behavior while arguing. Cornerstone for all these studies is the notion of *argument*. In particular, a great volume of those studies is devoted to give a definition about the internal structure of an argument, with the conspicuous majority to consent that it is essentially an inference from a set of statements (*premises*) to a central position (*conclusion*). Others are involved in studying the different interactions and correlations among arguments, while others view them from an abstract perspective in which arguments and relations form graphs. On top of these representations they propose reasoning methods for multiple purposes, like resolving conflicts, reaching common decisions, drawing conclusions etc. We noticed that some of the most prevailing concepts in the domain, are so discrete and compound, that they could also be included in the formalization of a data model for dialogical data. As a result, we suggest a data model consisting of graphs of structured arguments and relations, whereas ArgQL is designed according to these concepts. Note that the intention was to keep the model as abstract as possible, so that ArgQL will be compatible with the majority of the models in the area of Argumentation.

In its current form, ArgQL supports a number of information requirements, that can be classified to one (or more) of the following categories:

a. *Locating individual arguments*. Queries in this category express constraints on the internal structure of arguments. For example queries asking for arguments which justify a particular conclusion, requiring the existence of specific values in the premises of arguments, etc.

b. *Identifying commonalities in arguments' structure*. Queries in this category search for correlated arguments based on their content. For example, someone can ask for pairs of arguments the premise sets of which have common elements.

c. *Extracting argument relations*. ArgQL offers built-in keywords that allow the expression of restrictions about the relations between arguments. Depending on the different types of relations defined in the data model, the respective set of keywords is defined in the language to identify each type.

d. *Traversing the argument graph.* Within this category, ArgQL provides expressions used for navigation across the graph, which, combined with the mechanisms for filtering the argument structure, allow the identification of complete paths within the graph.

The syntax of ArgQL has been influenced by Cypher [48] and SPARQL [87], both being graph query languages, and it is based on the idea of pattern matching. This means that it provides expressions which are used as motifs in order to detect information matching to them. The formal definition of those patterns have been designed according to the data model and, therefore, we support patterns that restrict the internal structure of arguments, as well as patterns for graph navigation. ArgQL gives also special emphasis to the content equivalence, in the sense that it takes into account the fact that a specific content can be paraphrased in many different ways. This means whenever the search is restricted according to a particular content value, the query is also evaluated against all of its equivalent ones. The returning values of ArgQL can be either simple elements from the knowledge base, like arguments, propositions etc., or complete paths. Regarding the semantics, they are mainly influenced by the semantics of first-order logic. In particular, we define an *interpretation* function which is applied on all the different types of patterns and returns sets with all of the matching elements from the knowledge base.

One of the major objectives of this thesis was to showcase the feasibility of ArgQL in real application domains. To this end, we came up against the dilemma about whether we should pursue a native implementation and build our own methods and algorithms to execute the queries, or translate it in some already existing storage scheme. The first option would require us to deal with a number of problems already known for their complexity relevant to graph theory, like traversing and reachability issues, isomorphism, homomorphism etc. a task that falls out of the range of this thesis. Furthermore, it would require to generate data represented in the proposed data model. Therefore, we chose to leverage the maturity and efficiency of standardized storage schemes, by translating ArgQL into other query languages. In the current thesis, we show the case of RDF and the associated query language SPARQL. At this point, we need to emphasize that the process is not bound with this scheme, but it can be reproduced easily for other models as well, like First Order Logic, Cypher, Relational Databases etc. The reasons for choosing to show the particular storage scheme though, are manifold. First of all, the main target domain of ArgQL is the Web, where SPARQL constitutes the standard query language. In addition, a wide number of tools in the area of argumentation [57, 66, 88, 90, 92, 105] already extract their data in the particular format (AIF) and this way we took advantage of these datasets to test our queries. The implementation of the translation, verified our initial assertion (which also impelled our research), that, expressing the particular information requirements in the syntax of the standard languages could be a quite struggling task. On the other hand, the experimental evaluation that followed revealed that, translating ArgQL in a standard language was a good choice since, in most of the cases, the execution times were within satisfying limits, while in the cases that the execution was a bit slower, was due to limitations of the target language.

1.5 Contributions of this Dissertation

The key contributions of this thesis are the following:

- We propose a data model for representing dialogical data. It adopts abstract semantics being able to capture a multitude of theoretical models in the field of Argumentation, making the data model itself and by extension the query language compatible with them. To the best of our knowledge, this is the first time that semantics from the area of Argumentation are used in order to define a pure storage format for dialogical data.

- We propose a query language for argumentative data. ArgQL constitutes the first query language that formalizes in its specification the information requirements related with the domain of dialogues. We define the syntax both in a formal way and as an EBNF grammar and we also define the ArgQL semantics.

- We describe a mapping to the RDF storage scheme. In particular regarding the data models, we show the correspondence between each of the concepts of our data model and the concepts of the AIF ontology in RDFS. Concerning the translation into SPARQL, we present, in a formal way, the translation from each ArgQL pattern to a SPARQL graph pattern. We prove the correctness of the suggested translation and that it is semantics preserving, based on the semantics of the two languages and the suggested data and query mappings.

- Given that the proposed translation, in practice, does not generate the optimal possible queries, we suggest an optimization of the proposed translation which achieves shorter and by extension more efficient queries.
- We conduct an experimental study to evaluate the query execution and in particular, the translation along with its optimization. In this study we provide an estimation about the sizes of the generated queries, we show how the different components of the language affect not only that size but also the execution times, we evaluate the performance before and after the suggested optimization and finally, we measure the scalability in increasing data volumes.
- We provide an ArgQL endpoint, to allow for testing the language in a real dataset.

1.6 Outline of Dissertation

This thesis is organized as follows:

Chapter 2 provides some background knowledge that the reader will find useful in order to achieve a deeper comprehension of the related fields. Specifically, it offers a brief description of the most relevant concepts in the area of Argumentation. Then we present some preliminary terminology from RDF/s and SPARQL, which we will use to present the translation. Since there are no equivalent query languages to which we can compare ourselves, we also present in this section some models from the area of Argumentation that deal with the problem of retrieving information from argumentative data, but view it from a different research angle.

A comprehensive description of the language is given in chapter 3. In particular, in section 3.1 we describe the theory of the proposed data model. In section 3.2 we present the language informally by analyzing the informational requirements that currently covers, along with some query examples that enable an initial familiarization with the language. Then, in section 3.3 we give the formal specification of the language. The definition of the different patterns as well as its syntax are given in section 3.3.2, while the formal semantics of the language are presented in section 3.3.3.

Chapter 4 presents formally our methodology for query execution. In particular, in section 4.1.1 we give the mappings between our data model to the RDF format, while in section 4.1.2 we present the translation of ArgQL into SPARQL. The proof of correctness of the translation is given in the appendix .2. Section 4.2 describes some optimizations that concern the particular methodology.

Chapter 5 deals with technical issues that regard the execution and the efficiency of ArgQL. In section 5.1 we describe the implementation of the language. We present the basic algorithms at the parsing stage (Section 5.1.2), as well as those at the result stage, when the results returned by the SPARQL query are collected in order to create the final answer of an ArgQL query (Section 5.1.3). Furthermore, in section 5.1.4 we introduce the ArgQL endpoint as one of the outcomes of this research. A comprehensive experimental study is given in section 5.2.

Finally, some conclusions are drawn in chapter 6, which summarizes all that have been described in this document, and also includes discussion about future directions of this work.

Chapter 2
Background and Related Work

Contents

This chapter is devoted to some background knowledge and related work with respect to our research. In particular, section 2.1, provides a horizontal overview of the main aspects in the area of Computational Argumentation, since its representations and models highly influenced an important part of our work so far, but also the directions in which it can be extended in the future. We already mentioned that, ArgQL is the first query language targeting a data model for argumentation. Therefore, there are no similar languages with which we can directly compare ourselves. However, in the literature several approaches can be found that deal with the general problem of searching and extracting data from argumentative dialogues, but use standard models of other fields to address it. In section 2.2 we review some of these models. Finally, section 2.3 presents some preliminaries for the RDF/SPARQL languages since our data model and ArgQL are translated in them in chapter 4.

2.1 Argumentation

2.1.1 Overview

In this section we give a concise overview of the area of Argumentation and the research problems tackled by its theoretical and computational models. As a cognitive capacity, argumentation is important for handling conflicting beliefs, assumptions, viewpoints, opinions, goals, and many other kinds of mental attitudes. Faced with situations of incomplete of inconsistent information, people often resort to a process of constructing arguments in favor or against a given position, in order to make sense of the situation. Then, they engage in interactions with other people in order to exchange arguments in a cooperative or competitive attitude to reach a final agreement and/or to defend and promote an individual position. Exactly this procedure of reasoning is what the computational models of the argumentation area simulate. In particular, Argumentation promises powerful methods for reasoning with inconsistent knowledge and handling uncertainty [64, 80] in order to draw general conclusions

We borrow an image from [11] which gives an insightful illustration of the key aspects within a typical process of argumentation (Figure 2.1). There are five main stages that characterize this process: *structural, relational dialogical, assessment and rhetorical.* The boundaries between those layers usually are not tangible and some of the referred models may be involved in more than one stages. We will describe each layer separately, by giving some of the most representative argumentation frameworks for each stage. Regarding the first two stages, since they constitute the basic background of our work, will be discussed in more details the next section.

Structured layer. At this point arguments are constructed. Issues that concern the particular layer, are related with decisions about the internal structure of arguments in terms of their constitution and the relations between their ingredients. In all of the existing representations, arguments are created under a particular logic, and they essentially represent an inference using the method defined by this logic. Roughly, an argument can be seen as a pair $\langle \Phi, \alpha \rangle$, where Φ is a subset of the knowledge base (a set of formulae) that logically entails α (a formula). Here, Φ is called *support* and α is the claim. More about the different forms of an argument is given in the following section.

Relational layer. The relational layer deals with the identification and the formal representation of the different ways in which arguments interlink. Some of the most common types of relations are the a) *subargument/superarguemnt* relationships, indicating how an argument is built incrementally on top of other arguments, b) the *attack* relation that keeps all the different ways in which an argument may be withdrawn by other arguments, c) the *support* relation representing the ways that an argument supports another one by amplifying its conclusion or some of its premises and d) the *preference* which creates a ranking relation among arguments according to some criteria.

Dialogical layer. The dialogical layer deals with issues related to the process of argument exchange. That exchange is usually formalized as a game, which is made up of a set of communicative acts called *moves*, and is governed by a protocol, that is a set of rules that define the allowed moves

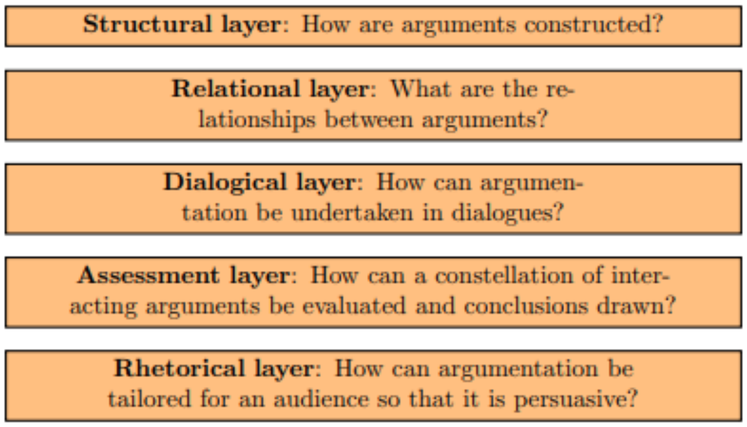

Structural layer: How are arguments constructed?
Relational layer: What are the relationships between arguments?
Dialogical layer: How can argumentation be undertaken in dialogues?
Assessment layer: How can a constellation of interacting arguments be evaluated and conclusions drawn?
Rhetorical layer: How can argumentation be tailored for an audience so that it is persuasive?

Figure 2.1: Key aspects of argumentation

at each stage of the dialogue. The protocol also specifies which must be the content of the locutions. Some of the models also formalize the outcome of the game, creating that way an intersection with the next layer: *assesment*. Douglas Walton in his book [120] makes a valuable categorization in the types of dialogues, based upon the information the participants have at the commencement of a dialogue, their individual goals for the dialogue and the goals they share. Some of these categories are: *Information-Seeking Dialogues, Inquiry Dialogues, Persuasion Dialogues, Negotiation Dialogues, Deliberation Dialogues etc.*

One of the most known dialogue games has been proposed by Prakken in [82], which formulates the protocol for a persuasion dialogue, having as a goal to resolve a conflict of opinion. This means that it is required from the dialogue to have a final outcome and therefore the approach captures also the next layer. Other equally significant dialogue games found in the literature are: [28] which focus on inquiry dialogue games, [96] that explore how argumentation schemes and their critical questions can be characterized as an extension to traditional dialectical systems and [4] where a general and abstract formal setting for argumentative dialogue protocols is proposed.

Assessment layer. In this layer, the created dialogue is evaluated in order to establish the justification status of the exchanged arguments. As we mentioned before, there are some argumentation frameworks that suggest evaluation methods over the defined structures of arguments and their relations. One of the most known such frameworks is the DELP (Defeasible Logic Programming) [51], an argumentation framework, the formalism of which combines results of Logic Programming and Defeasible Argumentation for warranting the entailed conclusions and deciding between contradictory goals. In this framework, the arguments and relations are explicitly defined, and on top of these formalism, they propose methods to evaluate the created dialectical trees, constituted by a number of argumentation lines. The Carneades model of argument [53] is a formal, mathematical model of argument structure and evaluation, taking seriously the procedural and dialectical aspects of argumentation. The model applies proof standards to determine the acceptability of statements on an issue-by-issue basis.

There is a large portion among the argumentation frameworks though, which do not rely on the structure of arguments to compute acceptance problems, but instead, they examine other external features for this purpose. Perhaps the most influential approach in the area of argumentation constitute the Abstract Argumentation Frameworks with the one of Dung [38] having paved the way to them. In abstract argumentation frameworks, the internal structure of arguments is indifferent, and dialogues are faced as graphs, the nodes of which represent arguments, but as abstract entities, and the edges represent the relations among them. Within such an arrangement, graph-oriented methods are applied to assess the acceptability of arguments. Due to their importance, we devote a separate section to describe these frameworks. Several extensions have been proposed that import other domain-related features in the evaluation process like the preference-based abstract argumentation [8], value-based argumentation frameworks [16], bipolar argumentation [33] etc.

Another significant category of frameworks that can be classified in the models of this stage, constitute the weighted argumentation frameworks. We refer to frameworks that assign weights on arguments (e.g. [5–7, 40]) or on attacks (e.g. [36, 41]) and draw conclusions about the strength of arguments based on those numbers. These weights may be some arbitrary number, but in some cases [61, 77] , they express the degree of arguments' uncertainty, and as a result, the evaluation of acceptance shifts to the probabilistic theory. Other factors that can affect the acceptability of an argument, are related with trust issues about the resource from which it originated, [76, 108], individual preferences [8, 72] etc.

Rhetorical layer. Normally argumentation is undertaken in some wider context of individual goals for the participants, which may be different from the common one. These goals reflect in the strategy in which they construct and communicate their argument, combined with whom will be the one that will receive it. In this stage, concepts like *persuasion*, *audience* and *Beliefs Desires Intentions (BDI)* are inserted in the reasoning models.

Formulations built around persuasion are based on the idea that, when two (or more) interlocutors disagree on a state of affairs, then the goal of each one when building an argument to communicate, is to make the rest change their minds and internalize this argument. Thus, he reasons not only about his own profile and knowledge, but also with respect to the profiles (or more precisely his perception about the profiles) of the audience. Some representative argumentation frameworks dealing which such issues follow.

In the Value-based Argumentation Frameworks (VAF) [16] they model persuasion as follows: Built on top of abstract argumentation, each argument promotes a set of social values. Audience is also modeled as an ordering between those values. The strength of an argument depends on the social values that it advances, and that whether the attack of one argument on another succeeds depends on the comparative strength of the values advanced by the arguments concerned. In [20, 59], on top of logic-based argumentation formalisms, the resonance (or impact) of an argument, depends on the extent to which the justification of the argument, agrees with what the audience regards as important, otherwise, to which extent, the justification and the beliefs of the audience intersect. In [9] they define three measures for analyzing dialogs from the point of view of an external agent: i) measures of the quality of the exchanged arguments in terms of their strengths, ii) measures of the behavior of each participating agent in terms of its *coherence*, its *aggressiveness* in the dialog, and finally in terms of the *novelty* of its arguments, iii) measures of the quality of the dialog itself in terms of the *relevance* and *usefulness* of its moves. Some additional persuasion models can be found in [29, 44, 82, 113].

BDI is a model, which is usually met in cognitive systems and is used to represent the mental state of intelligent agents. In particular, they can be represented by the following components: *Beliefs* represent the knowledge of the world, *Desires* represent the objective to accomplish and *Intentions* represent the course of actions currently under execution, in order to achieve the desires. Their methods find valuable application in the argumentation reasoning process since they may highly influence the way in which participants argue towards the satisfaction of their goals. Some examples of argumentative systems that use the BDI model are given in [78, 101, 102]

Argumentation in multi-agent environments

Multi-agent systems is a domain in which all those models of argumentation can offer valuable application. In essence, agents are independent and intelligent entities, with particular cognitive skills and the smooth coexistence in the same environment of more than one agents, requires that they are able to communicate with each other, in order to result in agreements when some issue arises. The motives that may bring them in such communication are manifold, like the resolution of a conflict, the joint decision making, the determination of some plan of actions (practical reasoning) etc. For all these problems, there is covering with all of the argumentation stages described above. A deeper analysis of this area would fall out of the scope of our research. For the interested reader though, in [91] there is a whole chapter devoted to the relation between argumentation and multi-agent systems.

2.1.2 Argument Representations

An argument is the elementary unit in a typical argumentation process. Especially in the early years of argumentation, many philosophers and experts in the area, invested efforts to give a precise but generic though answer to the question that raised much controversy in the community: ”What is an argument?”

Philosopher Stephen E. Toulmin [111] proposed argumentation model of law, in which (1) the proponent is trying to prove ***claim***, by giving (2) evidence or ***data*** to support that claim and (3) underlies the argument with assumptions or presuppositions that constitute the ***warrant***. These three notions constitute the required part. In addition, he defined an optional part that consists of the following: *qualifiers* give a qualitative measurement for the uncertainty of the argument, *rebuttals* give the points where an argument can be withdrawn and *backing* is used for extra evidence to the warrant. Figure 2.2 shows Toulmin’s original argument pattern. Although theoretical and intuitive enough, Toulmin’ s model has been many times considered as a reference point to the research of computational argumentation. Several extensions of his model have been proposed ([15], [12], [50])

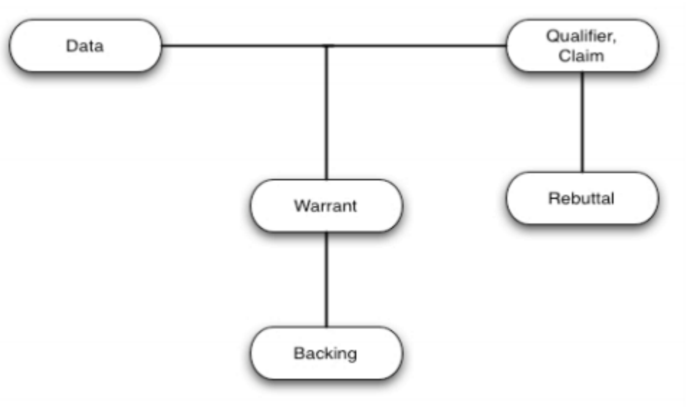


Figure 2.2: An interpretation of Toulmin's model

Walton [122] introduced twenty-five *argumentation schemes* namely, stereotypical, non-deductive and non-monotonic patterns of reasoning (structures of inference) able to represent arguments of our everyday life. *"Argument from Position to Know", "Appeal to Expert Opinion", "Argument from Cause to Effect"* are some of these patterns. Each scheme has a corresponding set of critical questions, representing its defeasibility conditions and the possible weak points that the interlocutor can use to challenge the argument. Below is an example for *argument from expert opinion*:

Major Premise: Source *E* is an expert in subject domain *S* containing proposition *A*.

Minor Premise: *E* asserts that proposition *A* (in domain *S*) is true(false)

Conclusion: *A* may plausibly be taken to be true (false)

The standard six basic critical questions matching the appeal to this scheme are the following.

1. *Expertise Question:* How credible is E as an expert source?
2. *Field Question:* Is E an expert in the field F that A is in?
3. *Opinion Question:* What did E assert that implies A?
4. *Trustworthiness Question:* Is E personally reliable as a source?
5. *Consistency Question:* Is A consistent with what other experts assert?
6. *Backup Evidence Question:* Is E's assertion based on evidence?

One of the most important points in these argumentation schemes, lies in their capability to offer complete arguments by filling missing statements from *enthymemes*, that is arguments whose premises or conclusion are either left unstated or are implied. Despite the lack of logical formality in the reasoning procedures and the interactions between arguments of these forms, Walton's schemes have significantly influenced the research especially in the areas of social argumentation and argument mining.

Despite the controversy around its constitution, there is a point of reference to which, most of the representations seem to conform, and it is that an argument is a pair $\langle Support, Conclusion\rangle$, such that the *Conclusion* part is inferred by the *Support*. The majority of the frameworks use formal languages to define their arguments around this general concept, and the differentiations are mainly detected in the inference method. One such model was given by Besnard and Hunter [20] in which arguments are defined in classical logic as

Definition 1. *An argument is a pair* $\langle \Phi, \alpha\rangle$ *such that*

- $\Phi \nvdash \bot$
- $\Phi \vdash \alpha$
- Φ *is a minimal subset of* Δ*, where* Δ *is a finite set of formulae.*

On top of this argument structure, three types of relations are defined:

Definition 2 (Defeater)**.** *A* ***defeater*** *for an argument* $\langle \Phi, \alpha\rangle$ *is an argument* $\langle \Psi, \beta\rangle$*, such that* $\beta \vdash \neg(\phi_1 \wedge \cdots \wedge \phi_n)$ *for some* $(\phi_1, \ldots, \phi_n) \subseteq \Phi$

Definition 3 (Undercut)**.** *An* ***undercut*** *for an argument* $\langle \Phi, \alpha\rangle$ *is an argument* $\langle \Psi, \neg(\phi_1 \wedge \cdots \wedge \phi_n)\rangle$*, where* $(\phi_1, \ldots, \phi_n) \subseteq \Phi$

Definition 4 (Rebuttal)**.** *An* ***rebuttal*** *for an argument* $\langle \Phi, \alpha\rangle$ *is an argument* $\langle \Psi, \beta\rangle$*, iff* $\beta \leftrightarrow \alpha$

Arguments do not represent proofs, instead their nature is defeasible. It is explained in terms of the interactions between conflicting arguments: inferences can be defeated and conclusions which have been drawn, may be later withdrawn, when additional information becomes available that gives rise to stronger counterarguments. Vreeswijk in [116] defined the arguments as "defeasible proofs" and used an unstructured language without logical connectiveness such as negation, to formalize them. His formalization was later used by various versions of ASPIC formalism [84]. In particular:

Definition 5. *An argument A on the basis of a knowledge base* $(\mathcal{K}, \leq)$ *in an argumentation system* $(\mathcal{L}, -, \mathcal{R}, \leq)$ *is:*

1. φ *if* $\varphi \in \mathcal{K}$ *with*
   $Prem(A) = \{\varphi\}$,
   $Conc(A) = \varphi$,
   $Sub(A) = \varphi$,
   $DefRules(A) = \varnothing$,
   $TopRule(A) = undefined.$

2. $A_1, \ldots, A_n \rightarrow \psi$ *if* $A_1, \ldots, A_n$ *are arguments such that there exists a strict rule*
   $Conc(A_1), ldots, Conc(A_n) \rightarrow \psi$ *in* $\mathcal{R}_s$,

$Prem(A) = Prem(A_1) \cup \ldots \cup Prem(A_n)$,
$Conc(A) = \psi$,
$Sub(A) = Sub(A_1) \cup \ldots \cup Sub(A_n) \cup \{A\}$,
$DefRules(A) = DefRules(A_1) \cup \cdots \cup DefRules(A_n)$,
$TopRule(A) = Conc(A_1), \ldots, Conc(A_n) \rightarrow \psi$

3. *$A_1, \ldots, A_n \Rightarrow \psi$ if $A_1, \ldots, A_n$ are arguments such that there exists a defeasible rule $Conc(A_1)$, $\ldots$, $Conc(A_n) \Rightarrow \psi$ in $\mathcal{R}_s$*
$Prem(A) = Prem(A_1) \cup \ldots \cup Prem(A_n)$,
$Conc(A) = \psi$,
$Sub(A) = Sub(A_1) \cup \ldots \cup Sub(A_n) \cup \{A\}$,
$DefRules(A) = DefRules(A_1) \cup \cdots \cup DefRules(A_n) \cup \{Conc(A_1), \ldots Conc(A_n) \Rightarrow \psi\}$,
$TopRule(A) = Conc(A_1), \ldots Conc(A_n) \Rightarrow \psi$

Other non-monotonic reasoning formalisms that have been proposed are described in ([80], [86]). Some works represent defeasibility as uncertainty in the premises e.g argumentation based on assumptions [39, 115], while others express it by using defeasible inference rules [51]. However, in [18, 19, 103], it has been shown that by having a classical logic-based notion of argumentation we can have a much deeper understanding of the individual arguments and of the counterarguments that impact on each argument. Some general reviews of formalisms for argumentation can be found in [47].

2.1.3 Abstract Argumentation Frameworks

According to Dung [38] an argumentation framework is defined as:

Theorem 1. *An argumentation framework (AF) is a pair (Arg, R) where*

- *Arg is a finite set of arguments*
- *$R \subseteq Arg \times Arg$ is a relation representing attacks.*

An example of a simple argumentation framework is depicted in figure 2.3

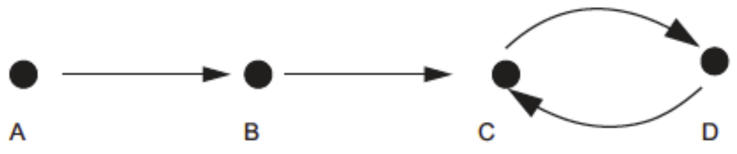


Figure 2.3: A simple Argumentation Framework [13]

In the graph of figure 2.3, A attacks B, B attacks C, and there is a mutual attack (or conflict) between C and D. These are called *directed attacks* and there is also the relation of *defense* between

A and C as A directly attacks the attacker of B, so indirectly it defends B. On top of this representation, he introduced a number of semantics that, essentially constitute different ”contexts” under which the status of arguments is determined. In the paper of Baroni et. al [13], it is asserted that an argument in a AF may be characterized by one of three statuses: accepted, rejected or undecided. This is expressed with the labelling $\Lambda = \{in, out, undec\}$ respectively and it constitutes a valuable effort to describe the semantics of Dung through these labels. An alternative approach for assigning labels can be found in [62] that use the set $\{+, -, ?\}$ for accepted, rejected and undecided respectively. Dung defined acceptability through the term of extension, that is subsets of *Arg* that include only acceptable arguments or with the labelling semantics, these arguments that are labeled as *in*. So, an argument, may or may not be included in an extension (there is not a sense of undefined argument). Before starting describing the semantics, it is worth introducing some definitions.

Theorem 2. *Given an Argumentation Framework AF = (Arg, R), a set $S \subseteq Arg$ is conflict-free, iff for each $a, b \in S, (a, b) \notin R$*

Conflict free is every set of arguments not attacking each other. From the example in figure 2.3, the conflict-free sets are $\{A\}, \{B\}, \{C\}, \{D\}, \{A, C\}, \{A, D\}, \{B, D\}$. Admissibility is defined based on the notion of defending arguments as follows:

Theorem 3. *Given an Argumentation Framework AF = (Arg, R), a set $S \subseteq A$ is admissible, iff S is conflict-free and $S \subseteq F(S)$, where $F(S) = \{A \mid S \text{ defends } A\}$*

Meaning that the arguments included in the set S are acceptable with respect to S. The admissible sets in figure 2.3 are $\{A\}, \{D\}, \{A, C\}$ and $\{A, D\}$. The following definition says that a complete set, includes all the arguments it defends, or otherwise defines the closure for admissibility.

Theorem 4. *Given an Argumentation Framework AF = (Arg, R), a set $S \subseteq A$ is complete, iff S is conflict-free and $S = F(S)$, where F(S) = $\{A \mid S \text{ defends } A\}$*

All complete sets are also admissible, however the opposite does not hold. For example the complete sets in the example are $\{A\}, \{A, C\}, \{A, D\}$. In the following definitions, we introduce the different semantics:

Theorem 5. *Given an Argumentation Framework AF = (Arg, R), the grounded extension of AF is a minimal with respect to set inclusion complete extension of AF.*

The grounded extension includes the arguments that are included in every extension and one cannot avoid to accept. It is not necessary that each AF has a grounded extension. In our example, there is one and it is the set $\{A\}$.

Theorem 6. *Given an Argumentation Framework AF = (Arg, R), the preferred extension of AF is a maximal with respect to set inclusion complete extension of AF.*

Preferred semantics express the idea to accept as many arguments as reasonably possible. In the example, two preferred extensions exist, namely $\{A,C\},\{A,D\}$.

Theorem 7. *Given an Argumentation Framework AF = (Arg, R), the stable extension of AF is a conflict-free set S and* $\forall a \in Args \setminus S$*, there exists* $b \in S$*, such that* $(a,b) \in R$

Stable semantics, indicate that no undecidable arguments exist in the graph, and they are either accepted or rejected. For the example in the figure, two stable extensions exist, namely $\{A,C\},\{A,D\}$.

In [13] they describe four alternative semantics: *semi-stable, ideal, stage and CF2* but we will not get into more details about them. Grounded semantics is the case of *skeptical* acceptance and can give solutions for problems that require a unique result. On the other hand, preferred and stable solutions give more than one results and is the case of *credulous* acceptance. In the example of figure 2.3, the different alternatives for each semantics, are caused because of the existence of a cycle in the graph, by considering each time one of the conflicted arguments as accepted and the other as defeated.

2.1.4 Argument Interchange Format (AIF)

The *Argument Interchange Format (AIF)* [34,89] has been devised in order to support the interchange of ideas and data between different projects and applications in the area of computational argumentation. In order to support such interchange, an abstract ontology for argumentation is presented, which serves as an interlingua between various more concrete argumentation languages. For example, a mapping from AIF to one of the most known argumentation frameworks, ASPIC+ [84] has been defined in [24].

The AIF core ontology is first and foremost an abstract, high-level specification of argumentative information and the relations of inference, conflict and preference between this information. The core ontology is intended as a conceptual model of arguments and the schemes or patterns arguments generally follow. It defines arguments and their mutual relations as typed graphs [92], which is an intuitive way of representing argument in a structured and systematic way without the formal constraints of a logic. Furthermore, this high-level description of the AIF ontology thus also meets [56]'s criterion of 'minimal encoding bias' for ontologies, which states that a conceptualization should be specified at the knowledge level, without depending on a particular symbol-level encoding. The high-level description of the AIF ontology can be made more concrete in individual specifications that make use of particular formalisms. Examples of such specifications of the AIF are the OWL-Description Logic specification proposed by [88], the RDFs specification as discussed by [92] and the Database Schema specification outlined in [89].

Description of the AIF Core Ontology

The AIF core ontology falls into two natural halves: the Upper Ontology and the Forms Ontology. The Upper Ontology defines the basic building blocks of AIF argument graphs, nodes and edges. The

Forms Ontology allows us to type the elements of AIF graphs in terms of argumentation-theoretical concepts such as inference, conflict and so on. Nodes can be used to build argument-graphs and these nodes then fulfill (i.e. instantiate) specific forms, such as inference schemes, from the Forms Ontology. In figure 2.4 an overview is given with the main description of the AIF ontology. White nodes define the classes (concepts) in the Upper Ontology whilst grey nodes define those in the Forms Ontology. The arrows denote the different relations between the classes in the ontology: dotted arrows are fulfills relations and normal arrows are is a (class inclusion) relations. So, for example, the class of inference schemes is a subclass of the class of schemes and RA-nodes (Rule Applications) fulfill inference schemes.

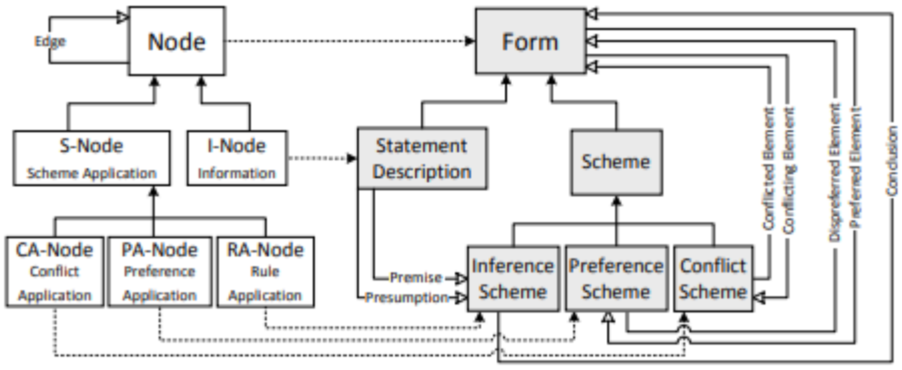


Figure 2.4: The AIF core ontology description. Dotted arrows are *fulfils* relations and normal arrows are *is a* relations (unless indicated otherwise)

The Upper Ontology places at its core a distinction between information, such as propositions and sentences, and applications of schemes, general patterns of reasoning such as inference or conflict. Accordingly, the Upper Ontology describes two types of nodes for building argument graphs: information nodes (I-nodes) and scheme application nodes (S-nodes), as well as an edge relation for the edges between nodes. S-nodes can be rule application nodes (RAnodes), which denote specific inference relations, conflict application nodes (CA-nodes), which denote specific conflict relations, and preference application nodes (PA-nodes), which denote specific preference relations. As [34] notes, nodes can have various attributes (e.g. creator, date). In the current AIF specification, a node consists of an identifier and some content (i.e. the information or the specific scheme that is being applied)

The Forms Ontology defines the schemes and types of statements commonly used in argumentation. The cornerstones of the Forms Ontology are schemes: inference, conflict and preference are treated as genera of a more abstract class of schematic relationships [26], which allows the three types of relationship to be treated in more or less the same way, which in turn greatly simplifies the ontological machinery required for handling them. Thus, inference schemes, conflict schemes and preference schemes in the Forms Ontology embody the general principles expressing how it is that q is inferable from p, p is in conflict with q, and p is preferable to q, respectively. The individual RA-, CA- and PA-nodes that fulfil these schemes then capture the passage or the process of actually inferring q from p, conflicting p with q and preferring p to q, respectively.

The nodes from the Upper Ontology can be used to build *AIF argument graph*, as follows:

Definition 6 (AIF graph)**.** *An AIF argument graph G_{AIF} is a simple digraph (V_{AIF}, E_{AIF}) where*

1. *$V_{AIF} = I \cup RA \cup CA \cup PA$ is the set of nodes in G_{AIF}, where I are the I-nodes, RA are the RA-nodes, CA are the CA-nodes and P A are the PA-nodes; and*
2. *$E_{AIF} \subseteq V_{AIF} \times V_{AIF} \setminus I \times I$ is the set of the edges in G_{AIF}. Any edge $e \in E_{AIF}$ is assumed to have exactly one type from among the following: premise, conclusion, preferred element, dispreferred element; and*
3. *if $v \in V_{AIF} \setminus I$ then v has at least one direct predecessor and one direct successor; and*
4. *if $v \in RA$ then v has at least one direct predecessor via a premise edge and exactly one direct successor via a conclusion edge; and*
5. *if $v \in PA$ then v has exactly one direct predecessor via a preferred element edge and exactly one direct successor via a dispreferred element edge; and*
6. *if $v \in CA$, then v has exactly one direct predecessor via a premise edge and exactly one direct successor via a conclusion edge.*

We say that, given two nodes $v_1, v_2 \in V_{AIF}$, v_1 is a *predecessor* of v_2 and v_2 is a *successor* of v_1 if there is a path in G_{AIF} from v_1 to v_2 and v_1 is a *direct predecessor* of v_2 and v_2 is a *direct successor* of v_1 if there is an edge $(v_1, v_2) \in E_{AIF}$. A node v is called an *initial* node if it has no predecessor.

Condition 2 states that I-nodes can only be connected to other I-nodes via S-nodes, that is, there must be a scheme that expresses the rationale behind the relation between I-nodes. S-nodes, on the other hand, can be connected to other S-nodes directly (see, e.g., Figures 3, 4). Condition 3 ensures that S-nodes always have at least one predecessor and successor, so that (a chain of) scheme applications always start and end with information in the form of an I-node.

Notice that in Definition 6 the edges are typed to indicate what the role of one node is with respect to another node. These edge types are defined in the Forms Ontology for each of the schemes and forms they connect (see the named arrows with the open heads in Figure 2.4). For example, a node that fulfils an inference or e conflict scheme can have a predecessor via a *premise* or *presumption* edge. Conditions 4 - 6 in Definition 6 state the specific constraints on edges in an argument graph: inference applications (RA-nodes) always have at least one premise and at exactly one conclusion (4), preference applications are always between two distinct nodes representing the preference' s preferred element and dispreferred element (5) and conflict applications always lies between exactly two nodes, both being predecessor and successor at the same time, since v is connected with two types of edges with each one, an incoming premise edge and an outgoing conclusion edge (6). Regarding condition (6) we make a convention here that if we want to force a conflict to be symmetric, we say that the involved in the conflict nodes are at the same time predecessor and successor of the node $v \in CA$ and they are both connected with *premise* and *conclusion* edges. In the other case that the conflict is asymmetric, it has the meaning of "attack".

Inference schemes in the AIF ontology are similar to the rules of inference in a logic, in that they express the general principles that form the basis for actual inference. They can be deductive (e.g. the inference rules of propositional logic) or defeasible (e.g. [121]'s argumentation schemes). One example of an inference scheme is that of Defeasible Modus Ponens [84, 85], of which the *premises* are the minor premise ϕ and the major premise $\phi \rightsquigarrow \psi$ (here, $\rightsquigarrow$ is a connective standing for defeasible implication) and the conclusion is ψ, where ϕ and ψ are meta variables ranging over well formed formulae in some language. Figure 2 shows an actual argument based on this scheme, represented in the AIF ontology. The scheme is indicated next to the RA-node ra2 representing the application of the scheme and the edges show their respective types.

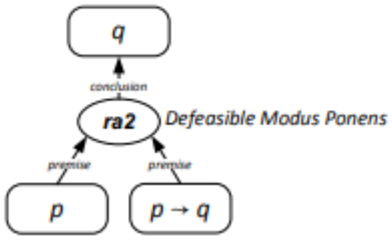


Figure 2.5: And AIF argument-graph

In figure 2.5, the fact that q is inferable from p is represented in the object layer as the (defeasible) conditional $p \rightsquigarrow q$. In line with a long tradition in argumentation theory and non-monotonic logic (e.g. [51, 99, 119], such specific knowledge can be modeled as inference rules itself, that is, as an inference scheme in the Forms Ontology. Take, for example, the inference scheme for Argument from Expert Opinion [119]:

Scheme for Argument from Expert Opinion

Premises: E is an expert in domain D, E asserts that P is true, P is within D;

Conclusion: P is true;

Presumptions: E is a credible expert, P is based on evidence;

Exceptions: E is not reliable, P is not consistent with the testimony of other experts.

An argument based on this scheme is rendered in Figure 2.6. Thus, specific (but still generalizable) knowledge can be modeled in the AIF in a principled way, using argumentation schemes, for which we can assume, for example, a raft of implicit assumptions which may be taken to hold and exceptions which may be taken not to hold. Note that the AIF ontology itself does not legislate which schemes are in the Forms Ontology and the exact structure of these schemes; rather, this depends on the inference rule schemes or argumentation schemes that a particular specification of the AIF ontology uses. Like inference, conflict is also generalizable. General conflict relations, which may be based on logic but also on linguistic or legal conventions, can be expressed as conflict schemes in the Forms Ontology.

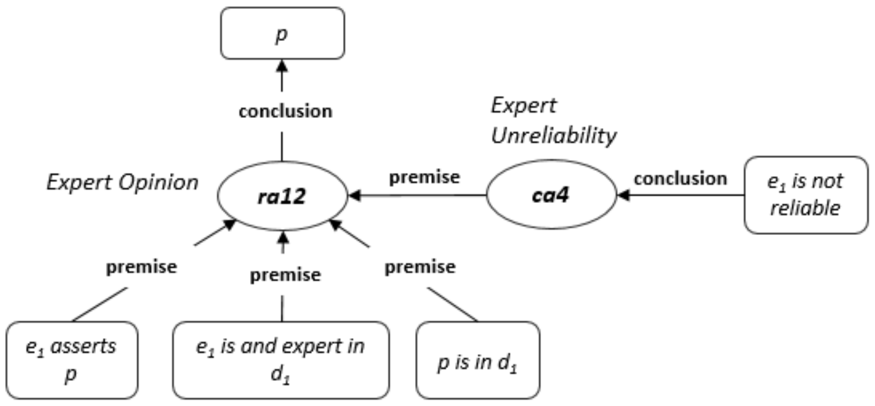


Figure 2.6: Conflict from Unreliability

As an example of a conflict scheme, take the scheme for Conflict From Expert Unreliability, which states that that the fact that an expert is unreliable is in conflict with the inference based on the Expert Opinion scheme. In other words, the conflicting element of this scheme is 'E is not reliable' and the conflicted element is the Scheme for Argument from Expert Opinion. Figure 2.6 shows the application of the conflict scheme Expert Unreliability, which here attacks the application of the Expert Opinion inference scheme as represented by the node ra12 (i.e. the fact that the expert e_1 is not reliable is in conflict with the fact that p is inferred from the premises).

AIF+

AIF+ [97, 98] is an extension of AIF used to represent dialogic argumentation. One of the challenges of AIF+ is to tie together the rules expressed in dialogue protocols with the inferential relations between premises and conclusions. The extensions are founded upon two important analogies which minimize the extra ontological machinery required. First locutions in a dialogue are analogous to AIF I-nodes, which capture propositional data. Second, steps between locutions are analogous to AIF S-nodes which capture inferential movement. The particular goals of AIF+ are listed bellow:

1. Extend the AIF to support representation of argumentation protocols (i.e. specifications of how dialogues are to proceed).

2. Extend the AIF to support representation of dialogue histories (i.e. records of how given dialogues did proceed).

3. Place little or no restriction on the types of dialogue protocol and dialogue history.
4. Integrate the dialogic argument representation of the AIF+ with the monologic argument representation of the AIF.
5. Meet all of 1-4 with the minimum extra representational machinery possible.

In this work, we do not take advantage of the entire conceptualization of AIF+, since the area of dialectical games, falls out of the scope of our research. We care exclusively about the content and the actual information of dialogues and not about a dialogue as an evolving process of information exchanging that confronts to a communication protocol (consisting of a set of particular types of movements (locutions) and a set of communication rules that define the behavior of an interlocutor in a dialogue), or about historical information. What we particularly need by this ontology, is the notion of *default rephrase*. Intuitively, a *default rephrase* represents the idea of paraphrasing and it holds between two propositions with the same or similar content expressed with different linguistic surface. Figure 2.7 represents a default rephrase between two I-nodes $i_1, i_2 \in I$.

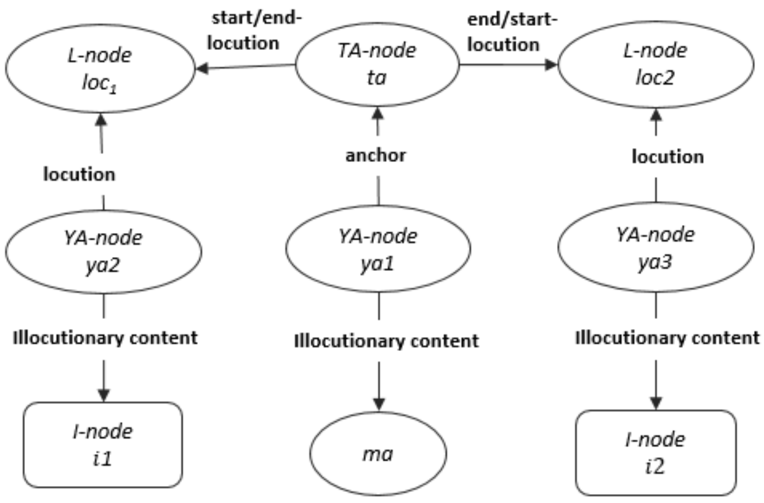


Figure 2.7: Default rephrase in AIF+ conceptualization

To represent this concept, AIF+ uses the following extra classes. L refers to the class Locution-node *(L-node)*. L-node inherits I-node and includes some extra meta-information that regard the existence of an I-node in the context of a dialogue, such as who said the proposition, when was it expressed, to which dialogue it is part etc. The aspect of dialogue is characterized as a form of inference called *transitional inference* and is represented by *TA-Nodes* (*TA*), Transition Application nodes, which capture the flow of a dialogue, for example recording that a given assertion has been made in response to an earlier question. *YA-nodes* (*YA*) are called *illocutionary schemes* and, roughly, describe the passage of a specific linguistic relation dependent on the type of illocutionary force used in speech act, or otherwise, the passage between L-nodes and I-nodes. Finally, *MA-nodes* (*MA*), capture the general concept of a *default rephrase*. Essentially, a default rephrase is represented as a transition from the dialectical content of the first I-node, to the respective dialectical content of the second. More about the AIF+ ontological extensions can be found in [97]. The V_{AIF} is now extended to include new classes as follows:

$V_{AIF+} = V_{AIF} \cup L \cup TA \cup YA \cup MA$

2.2 Argumentation in Social Web

With the emergence of the Social Web, the models of argumentation shifted towards the technologies of the web and since then, we witnessed an important increase in Web 2.0 human-centered collaborative deliberation tools. In only a few years of research, a huge number of argumentation tools appeared on the Web mostly aiming to facilitate the users' experience of online debating.

A comprehensive review of these tools can be found in [105]. There are various categories, to which these tools can be classified according to the functionality they provide. Some of the most common and important features are: querying, visualization, searching, evaluation, and user engagement. Since the number of these tools is quite big, we will not present here the extensive list, but will present a representative sample, by giving emphasis to the way in which they search for argumentative information within their data.

An interesting formalization of dialogues is provided by *Issue-Based Information Systems (IBIS)* [65], a problem-solving structure first published in 1970. As the name suggests, IBIS centers around controversial *issues*, which take the form of questions and it is especially intended to support community and political decision-making. It was originally designed as a documentation system, meant to organize discussion and allow subsequent understanding of the decision taken; this explains the use of "Information System" in its acronym. The context of the discussion is a *discourse* about a *topic*. Issues may bring up *questions of fact* and be discussed in *arguments*. Here, *"arguments"* are constructed in defense of or against the different positions until the issue is settled by convincing the opponents or decided by a formal decision procedure. IBIS also recognizes *model problems*, such as cost-benefit models, that deal with whole classes of problems.

The *Argumentation Markup Language (AML)* [2] is an XML interchange language for structured arguments. It is intended to be both human and machine readable, capable of representing many different forms of structured arguments. Its goals include the following three: 1) If tools can be said to be argumentation tools, then AML should be capable of representing their arguments 2) Argument viewing / browsing tools should be capable of displaying AML arguments that were developed using argumentation tools and 3) Argument editing tools should be capable of importing arguments, modifying them, and exporting the results, be the original arguments from the same or a different tool.

Araucaria [95] is a repository of arguments drawn from newspaper editorials, parliamentary reports and judicial summaries. It is chiefly intended for pedagogical use addressing the need to improve students' critical thinking. Using argumentation schemes, the result of any given analysis is a marked up version of the original text. That is, the text is interspersed with tags that indicate which parts of the text correspond to individual propositions, how these propositions relate to others, where particular argumentation schemes are instantiated, where enthymematic premises should be inserted and how a particular claim is evaluated by the analyst. The Araucaria System that creates files marked up according to AML is a tool of informal logic. In addition, there is not the notion of attack and conflicting arguments and as a result there is no mechanism that evaluates arguments according to their acceptability. As such, it can be employed as an aid to support argument analysis and as a diagrammatic presentation tool.

ArgDF [92] is a Semantic Web-based system, that uses the AIF-RDF ontology. ArgDF enables users to create and query arguments that are semantically annotated using different argumentation schemes. The system also allows users to manipulate arguments by attacking or supporting parts of existing arguments and also to re-use existing parts of an argument in the creation of new arguments. ArgDF also allows users to create new argumentation schemes. As such, ArgDF is an open platform not only for representing arguments, but also for building interlinked and dynamic argument networks.

Argunet [104] is a rather simple desktop tool for collaborative argumentation analysis by visualizing the structure of complex argumentations and debates, coupled with an open source federation system for sharing argument maps. A feature in which makes it unique is its multi-lingual environment. An Argument map is represented with multiple colors for better reading it. Arguments are reconstructed as premises and conclusions and there is also the notion of attack in the form of red or green arrows depending on whether it is a defeat or support relation respectively.

Avicenna is a Web-based system using Jena, ARQ and Pellet. The basic functionalities it offers, is firstly the chaining of arguments, that is retrieving all arguments that directly or indirectly support a given conclusion and secondly that arguments can be classified into the hierarchy of argument schemes defined by Walton.

Carneades [53] is a set of Open Source software tools providing support for a range of argumentation tasks including evaluation, construction and visualization. It is particularly suitable for legal applications. Users through a front-end web application, enter facts of cases to send feedback to legislative bodies about policy issues and legislative proposals. The legal norms and policies are modeled with high-level declarative rule language. The same language is also used to model Walton's argument schemes and critical questions, which are exploited for the classification of the arguments and for managing enthymemes. For the evaluation of statements during the dialectical process, four proof standards are used drawn from logic (dialectical validity) and from legal proof standards (a scintilla of evidence, a preponderence of the evidence, beyond a reasonable doubt). Carneades Argument Graphs have also been translated to the specification of Argument Interchange Format and vice versa [21].

DebateGraph [127] is a cloud-based Web 2.0 application that uses concept maps to explore topics and issues associated with them. It is a wiki debate visualization tool which has been adopted for use at the Kyoto climate change summit and is being tested by EU projects. It offers the opportunity to the public to collaborate to ”externalize, visualize, question and evaluate all of the considerations that any member thinks may be relevant to the topic at hand”. In a few words it offers a way for somebody to learn about, deliberate and decide on complex issues. DebateGraph affords several visualizations, which can also be embedded in other websites, encouraging users to add links to related webpages within graphs.

ConvinceMe [3] is an online debating environment where users participate in real debates. *Battles* are one-to-one debates between two users. The rest of the users vote for the arguments that are more convincing to them and the winner of the debate is the one who gets the most votes. No structure is defined to the arguments and the counterarguments but it is the only debating tool that inserted the process of voting.

DiscourseDB [1] is the online database of opinions and commentary. By making use of wiki technology it collaboratively collects opinions of journalists and commentators from various news sources, about current events in word’s politics, economy and other issues. Users can search for opinions for or against a position, according to a specific author or topic entering also the parameters of time, venue and so forth. Although it is a useful system for exploring opinions, it does not give the opportunity for an online debate between the users. Argumentation theory and technology in terms of structuring the opinions or evaluating the positions are not taken into consideration.

Opinions Space [46] is a software developed by UC Berkeley’s Center for New Media also designed to collect and visualize opinions on a variety of topics. Unlike DiscourseDB, Online Space ’s users are not afforded with a rich in its searching alternatives tool for opinions. Instead they are given the opportunity for real time deliberations in a way of declaring their agreement/disagreement with an opinion and stating their own positions regarding the topic. The notable point here is that they use sliders to express their sentiment to the specific opinion, entering the notion of strength on this interaction. The system then creates a map of opinions which is visualized as various sized points, each one representing a different perspective and with the larger points to map more popular perspectives. Neither the specific system uses a structure for the opinions and is mostly useful for visualization of real debates.

Parmenides [31, 32, 74] is an argumentation tool, that allows for a more effective participation of citizens in governmental decisions, by gathering the opinions of public about a proposal for a political action. It exploits two methods of argument theory: Argument schemes to structure proposals and Argumentation Frameworks to diagrammatically analyze the opinions submitted by users. Participants respond to a series of questions by asking them whether they agree or disagree as well as the points of disagreement regarding the position. They are also asked for alternative proposals. At the end their complete position is summarized before it is submitted. The system then analyzes these positions using argumentation techniques in favor of the government giving her a more clear perception of the public sentiment for the specific topic.

Rationale [114] is a commercial package allowing the diagramming and visualization of arguments; while its predecessor, Reason!Able, was designed for the education domain, Rationale is aimed at lawyers. Rationale facilitates the creation of ‘box-and-arrow’ argument maps, where users link premises to conclusions with boxes and arrows.

In a nominative presentation some more tools are *Archelogos*, *Arvina*, the online evaluation tool for *DELP*, *TOAST* (the online evaluation tool for the ASPIC+ framework [84], *CasAPI*, *Cohere*, *Argue tuProlog* etc.

2.2.1 Argument Search and Extraction

The extraction of argumentative structures from text gains increasing attention in recent years, especially with the emergence of machine learning technologies which found important application in the domain of NLP. Arguments created by models adopting those algorithms, as well as those created by the preexisted argumentation tools have been collected in argument corpora, upon which, search engines have been developed, enabling for their retrieval.

Two of the most known argument search engines are the *AIFdb* [1] and the *args.me* [2]. Both of them support keyword search. AIFdb [66] is built on top of an RDF database that uses the AIF-RDF representation. It returns a visual representation of the argument maps in which the particular keywords are found. *args.me* [117] allows for the composition of approaches to acquiring, mining, assessing, indexing, querying, retrieving, ranking and presenting arguments while relying on standard infrastructure and interfaces. The search engine relies on an initial, freely accessible index of nearly 300k arguments crawled from reliable web sources. The returned results are distinguished between those which are for or against the ”issue” expressed in the search. Others systems performing argument retrieval are the IBM’s Project Debater [100] and ArgumenText [106]. All those systems rely on some of the standard infrastructure to implement the search. None of them defines a formal language having been designed for their needs. In the same sense, other approaches can be found in [10, 14, 43, 73, 90, 112]

In [81] an interesting framework for ranking retrieved arguments based on the notion of *relevance* is presented. They assess relevance in three quality dimensions: *rhetorical, logical and dialectical*. *Rhetorical* quality includes notions of persuasive effectiveness, correct language, vagueness and style. An argument of high rhetorical quality is well-written and appealing to the audience. Logical quality refers to an argument’s structure and composition. An argument of high logical quality is based on acceptable premises and combines them in a cogent way to support the argument’s claim. Dialectical quality captures an arguments’ contribution to the discourse. An argument of high dialectical quality is useful to support cooperative decision making or to resolve a conflict.

[1]//www.aifdb.org/search
[2]https://www.args.me/

2.3 RDF Storage Scheme

2.3.1 RDF

Resource Description Framework (RDF) [49] is a meta-data model, where the universe of discourse is a set of *resources*. A resource is essentially anything that can have a unique Universal Resource Identifier (URI). Resources are described using *binary predicates*, which are used to form descriptions *(triples)* of the form *(subject, predicate, object)*: a subject denotes the described resource, a predicate denotes a resource's property and an object the corresponding property's value. The predicate is also a resource, while an object can be a resource or a literal value. We consider two disjoint and infinite sets **U, L**, denoting the URIs and literals, respectively.

Definition 7 (RDF triple and RDF graph)**.** *An RDF triple t is a tuple (s,p,o)* $\in$ ***U*** $\times$ ***U*** $\times$ ***(U*** $\cup$ ***L)****, where s, p and o are the subject, predicate and object, respectively. An RDF graph G is a set of RDF triples.*

The RDF Schema (RDFS) language [37] provides a built-in vocabulary for asserting user-defined schemas and ontologies in the RDF data model. In the schema level, it provides mechanisms for declaring the classes and the properties of the model, as well as the semantic topology they form, which is created by defining the *domain* and *range* classes in the specification of each property (predicates *[rdfs:domain], [rdfs:range]*). In addition, *subsumption* relationships among classes and properties are expressed with the RDFS *[rdfs:subClassOf]* and *[rdfs:subPropertyOf]* predicates, respectively. At the data level one can assert class individuals (instances) with the *instance of* relationships of resources, using the RDF predicate *rdf:type* [type].

2.3.2 SPARQL

Protocol and RDF Query Language (SPARQL) [52] has been established as the standard query language for RDF. In this work we deal with SPARQL 1.1, to leverage some extra features this version offers, compared to previous ones, like path expressions.

Let a set of variables, W and UWL, UW, UL the sets $U \cup W \cup L$, $U \cup W$ and $U \cup L$, respectively. The core fragment of SPARQL, is based on two main syntactical units, *triple pattern* and *graph pattern*. These are defined as follows:

Definition 8 (Triple pattern)**.** *A triple pattern tp is a triple* $(sp, pp, op) \in UV \times UV \times UVL$*, where sp, pp, op are a subject pattern, predicate pattern and object pattern, respectively.*

Definition 9 (Graph pattern)**.** *A graph pattern is defined by the following abstract grammar:*

$$gp \rightarrow tp \mid gp \text{ AND } gp \mid gp \text{ OPT } gp \mid gp \text{ UNION } gp \mid gp \text{ FILTER } expr$$

where AND, OPT, and UNION are binary operators that correspond to SPARQL conjunction, OPTIONAL and UNION constructs, respectively. FILTER expr represents the FILTER construct with a boolean expression expr, which is constructed using elements of the set UVL*, constants, logical connectives* $(\neg, \wedge, \vee)$*, inequality symbols* $(<, \leq, >, \geq)$*, the equality symbol* $(=)$*, unary predicates and other features defined in [52]. Function var(gp) returns the set of variables that appear in gp.*

The main body of a SPARQL query is given by the following syntax:

Definition 10 (SPARQL query)**.** *A SPARQL query sparql is defined as:*

sparql → Select varlist Where gp

where varlist = ($v_1, \ldots, v_n$) is a set of variables and varlist ⊆ var(gp).

According to [79], the semantics of SPARQL are defined in terms of the following mapping function:

Definition 11 (Variable mapping)**.** *Let a mapping $\sigma : W \to UL$ be a partial function that assigns RDF terms of an RDF graph, to variables of a SPARQL query.*

Abusing notation, for a triple pattern tp we denote by $\sigma(tp)$ the triple obtained by replacing the variables in tp according to σ. The domain of σ, *dom(σ)*, is the subset of W over which σ is defined. Two mappings σ_1 and σ_2 are *compatible*, (written as $\sigma_1 \sim \sigma_2$), if $\sigma_1(x) = \sigma_2(x)$ for all variables $x \in dom(\sigma_1) \cap dom(\sigma_2)$. Mappings with disjoint domains are always compatible.

Let Ω_1 and Ω_2 be sets of mappings. The following operators (join, union, difference and left outer join) are defined between Ω_1 and Ω_2:

$$\Omega_1 \bowtie \Omega_2 = \{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in \Omega_1, \sigma_2 \in \Omega_2 \text{ are compatible mappings}\},$$
$$\Omega_1 \cup \Omega_2 = \{\sigma \mid \sigma \in \Omega_1 \text{ or } \sigma \in \Omega_2\},$$
$$\Omega_1 \setminus \Omega_2 = \{\sigma \in \Omega_1 \mid \text{for all } \sigma' \in \Omega_2, \sigma \text{ and } \sigma' \text{ are not compatible}\},$$
$$\Omega_1 :\bowtie \Omega_2 = \{(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}.$$

The evaluation of a SPARQL graph pattern *gp*, over an RDF graph *G*, is denoted as $\varepsilon_G(gp)$ and is defined in a recursive way in the next list.

- If *gp* is a triple pattern *tp*, then $\varepsilon_G(gp) = \{\sigma \mid \text{dom}(\sigma) = \text{var}(tp) \text{ and } \sigma(tp) \in G\}$, where var($tp$) is the set of variables occurring in *tp* and $\sigma(tp)$ is the triple obtained by replacing the variables in *tp* according to σ
- If *gp* is gp_1 *AND* gp_2, then $\varepsilon_G(gp) = \varepsilon_G(gp_1) \bowtie \varepsilon_G(gp_2)$
- If *gp* is gp_1 *OPT* gp_2, then $\varepsilon_G(gp) = \varepsilon_G(gp_1) :\bowtie \varepsilon_G(gp_2)$
- If *gp* is gp_1 *UNION* gp_2, then $\varepsilon_G(gp) = \varepsilon_G(gp_1) \cup \varepsilon_G(gp_2)$

The semantics of the FILTER expression *expr* is defined as follows. Given a mapping σ and an expression *expr*, σ satisfies *expr*, denoted by $\sigma \models_S expr$, iff:

expr is bound(?X) and ?X ∈ dom(σ);
expr is ?*X* op ł, ?X ∈ dom(σ) and σ(?X) op ł, where op → < | ≤ | ≥ | > | =;
expr is ?X op ?Y, ?X,?Y ∈ dom(σ) and σ(?X) op σ(?Y), where op → < | ≤ | ≥ | > | =;
expr is ($\neg expr_1$) and it is not the case that $\sigma \models_S expr_1$;

$expr$ is $(expr_1 \vee expr_2)$ and $\sigma \vDash expr_1$ or $\sigma \vDash_S expr_2$;
$expr$ is $(expr_1 \wedge expr_2)$, $\sigma \vDash expr_1$ and $\sigma \vDash_S expr_2$;

We extend the σ mapping so that we get the set of triples that result from the application of σ on a graph pattern gp, as follows:

- If gp is a triple pattern tp, then $\sigma(gp) = \{\sigma(tp)\}$
- If gp is any of the $(gp_1\ AND\ gp_2)$, $(gp_1\ OPT\ gp_2)$ or $(gp_1\ UNION\ gp_2)$ then $\sigma(gp) = \sigma(gp_1) \cup \sigma(gp_2)$

Now, for a given gp, we define the set $\mathcal{E}_G(gp)$ as the set that contains all the sets of matching triples for gp. In particular:
$\mathcal{E}_G(gp) = \{\sigma(gp) \mid \sigma \in \varepsilon_G(gp)\}$

Chapter 3
ArgQL - Formal Specification

Contents

In this chapter, we present the formal specification for the main core of our work. In particular, in the first section 3.1 we present the data model according to which argumentative data are structured. The data model was designed, based on the most prevailing concepts in the area of argumentation. The next section 3.2 presents the language informally by describing the categories of queries it allows to express, and through a number of representative queries for each category based on a data example. Later in this chapter, (section 3.3) we give the formal description of the language. In particular, in section 3.3.2 we present formally the main constructs of the language and its syntax, while in section 3.3.3, we give the semantics.

3.1 Data model in Structured Argumentation

The formal principles that rule the target data come from the theory of structured argumentation. We adopt Alfred Tarski's [109] abstract logic, who defined a general theory, of which, almost all well-known monotonic logics (classical logic, modal logic etc.) can be considered special cases. In particular, we assume the pair $\mathcal{L} =< \mathcal{P}, Cn >$, where $\mathcal{P}$ a set of well-formed formulas, called *propositions* and Cn is a function, mapping sets of propositions to sets of propositions *(consequence operator)*. The intuitive meaning of Cn is that a set X implies exactly the propositions contained in $Cn(X)$. Cn satisfies the following axioms:

1. $X \subseteq Cn(X)$ *(Expansion)*

2. $Cn((Cn(X))) = Cn(X)$ *(Idempotence)*
3. If $X \subseteq Y$, then $Cn(X) \subseteq Cn(Y)$ *(Monotonicity)*
4. $Cn(\varnothing) \neq \mathcal{P}$ *(Coherence)*

The following properties arise from the axioms above:

1. $Cn(X) = \bigcup_{Y \subseteq X} Cn(Y)$ *(Finiteness)*
2. If $Y \subseteq Cn(X)$ and $Z \subseteq Cn(X \cup Y)$, then $Z \subseteq Cn(X)$ *(Transitivity)*

The consequence operator allows to define the concepts of *equivalence* and *conflict* as follows:

Definition 12 (Equivalence)**.** *We say that two sets $X, Y \subseteq \mathcal{P}$ are equivalent (denoted by $X \equiv Y$) if and only if $Y \subseteq Cn(X)$ and $X \subseteq Cn(Y)$.*

Definition 13 (Conflict)**.** *We say that two sets $X, Y \subseteq \mathcal{P}$ are conflicting (denoted by $X \not\sim Y$) if and only if $Cn(X \cup Y) = \mathcal{P}$.*

In this work, we will need to express equivalence and conflict between single propositions, apart from sets of propositions. Thus, we abuse notation for the operators $\not\sim$ and $\equiv$ and will write, e.g. $x \not\sim y$, instead of $\{x\} \not\sim \{y\}$ for $x, y \in \mathcal{P}$.

We impose the following obvious propositions regarding the conflict and equivalence relations. The first one states that, there is no case two sets of propositions being equivalent and conflicting, at the same time. The second one postulates, that the conflicting of a proposition, is also conflicting to all of its equivalents.

Proposition 1. *For $X, Y \subset \mathcal{P}$, it cannot hold $X \not\sim Y$ and $X \equiv Y$ at the same time.*

Proposition 2. *For $X, Y, Z \subset \mathcal{P}$, If $X \not\sim Y$ and $Y \equiv Z$, then $X \not\sim Z$.*

The notion of consistency is defined as:

Definition 14 (Consistency)**.** *A set $X \subseteq \mathcal{P}$ is inconsistent with regard to the logic $\mathcal{L}$ iff $Cn(X) = \mathcal{P}$. Otherwise, it is consistent.*

Based on the consequence operator, we define an argument as a special case of inference, in which the inferred set is a singleton. A set of zero or more propositions justify the validity of that central proposition. Formally:

Definition 15 (Argument)**.** *Given a set of argument identifiers N, an argument of the logic $\mathcal{L}$ is a tuple (id, Pr, c), where*

- *$id \in N$ is the identifier of the argument*

- *$Pr \subseteq \mathcal{P}$,it is consistent and it is called premise*
- *$c \in Cn(Pr)$ and it is called conclusion and*
- *$\nexists\, pr \subset Pr$, s.t., $c \in Cn(pr)$.*

Given an argument a, we will refer with $id(a)$ to its identifier, with *prem(a)* to its premise set and with *concl(a)* to its conclusion.

The coexistence of the notions of conflict and equivalence in our language cause bipolar interactions between arguments, as well. Thus we discern two positive (support) and two negative (attack) types of relations defined as follows:

Definition 16 (Argument relations)**.** *We define two types of attack (rebut, undercut) and two types of support (endorse, backing) as:*

- *There is a rebut between two arguments a_1, a_2, iff $concl(a_1) \not\leftrightarrow concl(a_2)$*
- *There is an undercut from argument a_1 to a_2, iff $concl(a_1) \not\leftrightarrow p$ for $p \in prem(a_2)$*
- *There is an endorse between two arguments a_1, a_2, iff $concl(a_1) \equiv concl(a_2)$*
- *There is a back from argument a_1 to a_2, iff $concl(a_1) \equiv p$ for $p \in prem(a_2)$*

A knowledge base that confronts to the data model defined in this section, is called *argument base* and is defined as follows:

Definition 17 (Argument base)**.** *Given the logic $\mathcal{L}$, we define an argument base A as an arbitrary, but fixed set of structured arguments defined in $\mathcal{L}$. For an argument base A, we extract the following sets:*

· *$P(A) = \bigcup_{a \in A}(prem(a) \cup concl(a))$ is the logical content of the arguments in A.*

· *$cf(A) = \{(p_1, p_2) \mid p_1 \not\leftrightarrow p_2$ with $p_1 \in P(A)$ or $p_2 \in P(A)\}$ is a set of conflicts in $P(A)$*

· *$eq(A) = \{(p_1, p_2) \mid p_1 \equiv p_2$ with $p_1 \in P(A)$ or $p_2 \in P(A)\}$ is a set of equivalences in $P(A)$*

In an abstract level, arguments together with the relations of definition 16 form a graph. Intuitively such a graph of arguments represents the notion of structured dialogue and its definition is given below:

Definition 18. *Given an argument base A we define the argument graph $G_A = (A, R)$, in which A is a finite set of structured arguments and $R \subseteq A \times A$ the set of their relations, which can be of four different types $\{$rebut, undercut, endorse, back$\}$, depending on the existing interactions between the arguments.*

Like any graph model, the notion of *path* is a foreground concept. It allows to discuss about connectivity and reachability between the nodes. In the following, we define a path as:

Definition 19 (Path)**.** *Given an argument graph $G_A = (A,R)$, we define a path between two arguments a_1,a_n as the sequence $P_{a_1 \to a_n} = (a_1 \dots a_n)$ with $a_1,a_2,\dots,a_n \in A$ for which, $\exists r_i \in R$, such that $r_i = (a_i,a_{i+1})$, for $1 \leq i \leq n-1$. It also holds that $P_{a\to a} = a$.*

Definition 20 (Path concatenation)**.** *Given two paths $P_{a_1 \to a_x} = (a_1 \dots a_x)$ and $P_{a_x \to a_n} = (a_x \dots a_n)$, we denote their concatenation as $P_{a_1 \to a_x} \cdot P_{a_x \to a_n} = (a_1 \dots a_x \dots a_n) = P_{a_1 \to a_n}$*

Now, we will show the transition from the formal principles of the proposed data model, to the practical issues that concern their realization in actual data. In particular, we need to specify the syntax of the data, which will later define the expected answer to the queries.

At first, we adopt the quoted (string) representation for the proposition values. We use the angles $\langle \cdot \rangle$ to enclose the whole argument. The set of premises is inserted within the $\{\cdot\}$ brackets, while the conclusion is a single proposition after the comma. In the following we give an example of data that conform to the proposed data model. Note that wherever in this document we use labels on the arguments, it is for presentation purposes since it is is not part of the definition 15.

Example 3.1.1. *Let an argument base consisting of the following set of arguments A:*

$a : \langle \{a_1,a_2,a_3\},c_a \rangle$, *with* $a_1,a_2,a_3,c_a \in \mathcal{P}$ *and* $c_a \in Cn(\{a_1,a_2,a_3\})$

$b : \langle \{b_1\},c_b \rangle$, *with* $b_1,c_b \in \mathcal{P}$ *and* $c_b \in Cn(\{b_1\})$

$c : \langle \{c_1,c_2\},c_c \rangle$, *with* $c_1,c_2,c_c \in \mathcal{P}$ *and* $c_c \in Cn(\{c_1,c_2\})$

$d : \langle \{d_1,d_2\},c_d \rangle$, *with* $d_1,d_2,c_d \in \mathcal{P}$ *and* $c_d \in Cn(\{d_1,d_2\})$

$e : \langle \{e_1\},c_a \rangle$, *with* $e_1,c_e \in \mathcal{P}$ *and* $c_e \in Cn(\{e_1\})$

$f : \langle \{f_1,c_1\},c_f \rangle$, *with* $f_1,c_1,c_f \in \mathcal{P}$ *and* $c_f \in Cn(\{f_1,c_1\})$

$g : \langle \{g_1\},c_g \rangle$, *with* $g_1,g_2,c_g \in \mathcal{P}$ *and* $c_g \in Cn(\{g_1\})$

We also assume the set of conflicts between the involved propositions:

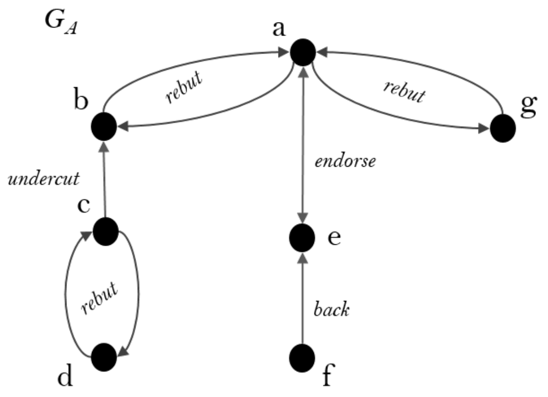
$cf(A) = \{(c_a,c_b),(c_c,b_1),(c_d,c_c),(c_g,c_a)\}$

and the set of equivalences:

$eq(A) = \{(c_f,e_1),(c_a,prop_1),(e_1,prop_2)\}$, *with* $prop_1,prop_2 \in \mathcal{P}$

For simplicity we did not use the quoted representation for the propositions. The next figure depicts the argument graph generated by A.

The relations between the arguments are created according to the definition 16 and the given set of conflicts and equivalences. Note that the rebut and endorse relations are symmetric and this happens because of the property of symmetry of the conflict and equivalence relations. On the contrary, the back and undercut relations are non-symmetrical.



3.2 ArgQL Description

In this section we provide a thorough demonstration of ArgQL and its main features. As we already mentioned, it constitutes a declarative language targeting data, which are structured according to the data model suggested in the section above. In particular, it targets both structured view of an argument base A (definition 17) as well as its abstract graph view G_A (definition 18). Currently, ArgQL provides mechanisms for retrieving data as they are saved in the knowledge base, whereas it does not allow modifications or reasoning about them.

The information requirements that a query language for structured dialogues should satisfy can be various from simple to more complex ones. In a first attempt to formally define such a language, we restrict ourselves to a specific set of requirements in which data are queried according to their structure and content. As a result, we currently support the following four general categories of queries: *a)* Locating individual arguments, *b)* Identifying commonalities in arguments' structure, *c)* Extracting argument relations and *d)* Traversing the argument graph. Among these categories, queries in *(a)* and *(b)* target the argument base A, while *(c)* and *(d)* target the G_A. Of course, ArgQL support hybrid queries, belonging to more than one categories. Below we discuss separately each of these categories and we give representative query examples against data of the example 3.1.1, to show how they are supported by the proposed language. In this section, we give an informal description of the language, while its formal syntax and semantics is presented in section 3.3.

Locating individual arguments. One of the main features of ArgQL is that it allows to identify specific arguments, using patterns and filters that restrict their internal structure. To do this, ArgQL provides a construct, called *argument pattern*, that matches with whole arguments. An argument pattern can be either a single variable or have the form ⟨*premise_pattern, conclusion_pattern*⟩. More about the concrete syntax of argument patterns can be found in the next section.

The simplest query in ArgQL is the one that asks for all arguments in the knowledge base. There are two ways to express this:

Q_1. **match** ?arg **return** ?arg

where the argument pattern has the form of a single variable ?arg which binds all arguments, or:

Q_2. **match** ?arg:⟨ ?pr, ?c ⟩

return ?arg

where variable ?pr binds the premises of all the arguments, variable ?c their conclusion and ?arg their complete value. The results are the same for both queries and these are:

1. [?arg: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$*]*
2. [?arg: (b) $\langle\{ b_1\}, c_b \rangle$*]*
3. [?arg: (c) $\langle\{ c_1, c_2\}, c_c \rangle$*]*
4. [?arg: (d) $\langle\{ d_1, d_2\}, c_d \rangle$*]*
5. [?arg: (e) $\langle\{ e_1\}, c_a \rangle$*]*
6. [?arg: (f) $\langle\{ f_1, c_1\}, c_f \rangle$*]*
7. [?arg: (g) $\langle\{ g_1\}, c_g \rangle$*]*

Other cases of individual arguments retrieval are queries which restrict the internal structure. For example the following query asks for arguments with the conclusion the proposition c_f:

Q_3. **match** ?arg: ⟨ ?pr, 'c_f' ⟩ **return** ?arg

for which the answer is

1. [?arg: (g) $\langle\{ g_1\}, c_g \rangle$*]*

The following query requires arguments for which the proposition c_1 exist in their premises:

Q_4. **match** ?arg: ⟨ ?pr[/{'c_1'}], ?c ⟩

return ?arg

for which the answer is:

1. [?arg: (c) $\langle\{ c_1, c_2\}, c_c \rangle$*]*
2. [?arg: (f) $\langle\{ f_1, c_1\}, c_f \rangle$*]*

With ArgQL we can also ask whether specific arguments are stored in the knowledge base. For example, the query:

Q_5. **match** ?arg: ⟨{ 'd_1', d_2'}, 'c_d' ⟩

return ?arg

will return the argument (d) as long as it exists in the knowledge base, or the empty set, otherwise.

Identifying commonalities in arguments' structure. ArgQL provides built-in mechanisms to identify sets of arguments with commonalities in their structure, and basically this feature concerns the premise part of arguments. Presently, it allows to restrict the premise part of an argument with regard to the premise part of another argument. For example, the following query asks for pairs of arguments with intersected premises, that is arguments with at least one common premise:

Q_6. **match** ?arg$_1$: ⟨ ?pr$_1$, ?c$_1$ ⟩, ?arg$_2$: ⟨ ?pr$_2$[.?pr$_1$], ?c$_2$ ⟩

return ?arg$_1$, ?arg$_2$

which gives the following two answers:

1. [?arg$_1$: (c) $\langle\{ c_1, c_2\}, c_c \rangle$, ?arg$_2$: (f) $\langle\{ f_1, c_1\}, c_f \rangle$]
2. [?arg$_1$: (f) $\langle\{ f_1, c_1\}, c_f \rangle$, ?arg$_2$: (c) $\langle\{ c_1, c_2\}, c_c \rangle$]

Note that these two answers are essentially the same but contain the arguments in different order. This happens because in the first answer, the argument *(c)* matches the first argument pattern and the argument *(f)* matches the second argument pattern, while in the second answer, *(f)* matches the first argument pattern and *(c)* matches the second.

Another case that falls in this category, is for example the restriction that the premises of an argument must be a subset of the premises of another. At a later time this feature can be extended in various ways, like searching for relevant arguments or even for percentage of relevance, based on a predefined theoretical model for its computation.

Extracting argument relations. In this current category, we ask to identify relations among arguments, which implicitly exist in the equivalences and conflicts among the propositions that compose the arguments, according to the definition 16. For this reason, the syntax provides four keywords *rebut, undercut, endorse, back* to allow for identifying the respective relations, as well as the general keywords *attack, support* that allow to leave the particular relations unspecified (e.g. with the *attack* keyword we ask either for *rebut*, or for *undercut*). For example the following query asks for arguments that support those with conclusion c_a, with the *endorse* type of relation:

Q_7. **match** ?arg$_1$ *endorse* ?arg$_2$: ⟨ ?pr$_2$, 'c_a' ⟩

return ?arg$_1$

This query has two answers:

1. [?arg$_1$: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]
2. [?arg$_1$: (e) $\langle\{ e_1\}, c_a \rangle$]

In this query arguments *a* and *e* are returned since they both have the same conclusion c_a. Note that Q_6 has the same results with the one that asks for arguments with conclusion c_a (case q_3) and this is due to the nature of the *endorse* relation, which is created when two arguments have the same conclusion. The same results would also be returned in case we had used the keyword *support* instead of the *endorse*. In this case, if there was a *back* relation between a different argument, let *x* and argument *a*, then *x* would also be included in the answers. Similar rationale holds on the attack relations, as well.

Traversing the argument graph. The last, but one of the most useful features of ArgQL, is the ability to navigate across the graph of arguments. In particular, it provides expressions that act as path motifs and, combined with the mechanisms for filtering the argument structure, allow to identify complete paths within the graph and return subgraphs as answers. For example, the following query returns arguments that are at a distance of 3 attack relations from an argument with conclusion 'c_a':

Q_8. **match** ?arg$_1$ *attack/attack/attack* ?arg$_2$: $\langle$?pr$_2$, 'c_a' $\rangle$

return ?arg$_1$

Query Q_8 gives the results:

1. [?arg$_1$: (d) $\langle\{ d_1, d_2\}, c_d \rangle$]
2. [?arg$_1$: (b) $\langle\{ b_1\}, c_b \rangle$]
3. [?arg$_1$: (g) $\langle\{ g_1\}, c_g \rangle$]

For expressions of the form `attack/attack/attack`, we can also use the shorthand *(attack)*3*. Note that the results include the arguments *(b)* and *(g)*, which, at first sight, are at distance 1 from argument *(a)*. This happens because of the bidirectional *rebut* relation, which cause a cycle between the involved arguments. More precisely, any path pattern of this type, with an odd number as a length indicator will give an answer which will include these two arguments.

There may be cases, where the desired path should be of variable length. For example we may need to express the requirement for paths of length *at least* 1, meaning paths of length 1, of length 2 and so on. On the other hand, the way in which argument relations have been defined, create graphs which may contain cycles of two or more arguments. Inside a cycle, the length of paths is undetermined, having as a consequence, such requirements like the one described here, to fall into situations of infinity and execution failure. Therefore, that feature has been designed in a way that warrants their execution in a limited time. In particular, with ArgQL we can ask for paths with length *at most* n, using the expression $+n$. In this way, regardless of the existing cycles and how many times the arguments within the same cycle will be visited, the execution will terminate, when the length of the matching path reaches the number n. For example the following query returns arguments which lie in distance of at most 3 attack relations from arguments with conclusion c_a.

Q_9. **match** ?arg$_1$ *(attack)+3* ?arg$_2$: $\langle$?pr$_2$, 'c_a' $\rangle$

```
    return ?arg1
```

and the answer to this query is:

1. [?arg_1: (b) $\langle\{ b_1\}, c_b \rangle$]
2. [?arg_1: (c) $\langle\{ c_1, c_2\}, c_c \rangle$]
3. [?arg_1: (d) $\langle\{ d_1, d_2\}, c_d \rangle$]
4. [?arg_1: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]
5. [?arg_1: (g) $\langle\{ g_1\}, c_g \rangle$]

Arguments (b) and (g) are at distance of 1 and 3 attacks from argument (a), argument (c) is at distance of 2, argument (d) is at distance of 3 while argument (a) is at distance of 2 attacks from itself.

Although Q_9 returns all matching arguments, it does not give any further information about the actual matching paths, like their length for each of the returning arguments or the intermediate arguments and relations. To address this, ArgQL offers the capability to ask for the complete matching path as an answer. To do this, we introduce the keyword *path* in the list of return values. We rewrite Q_9, such that it returns the complete path as an answer:

```
Q'9. match ?arg1 (attack)+3 ?arg2:  ⟨ ?pr2, 'ca' ⟩

    return path(?arg1, ?arg2)
```

The keyword *path* indicates a function, the input of which is the two variables used for the argument patterns on the left and right side of the same path motif. Any different input will generate parse error. Due to the existence of cycles, the particular path pattern matches with a wide number of paths in the data. Below we show an excerpt of the answer including a subset of the matching paths:

1. [*?arg_1: (b) $\langle\{ b_1\}, c_b \rangle$* - ATTACK - *?$arg_2$: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]*
2. [*?arg_1: (g) $\langle\{ g_1\}, c_g \rangle$* - ATTACK - *?$arg_2$: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]*
3. [*?arg_1: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$* - ATTACK - *_: (g) $\langle\{ g_1\}, c_g \rangle$* - ATTACK - *?$arg_2$: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]*
4. [*?arg_1: (d) $\langle\{ d_1, d_2\}, c_d \rangle$* - ATTACK - *_: (c) $\langle\{ c_1, c_2\}, c_c \rangle$* - ATTACK - *_: (b) $\langle\{ b_1\}, c_b \rangle$* - ATTACK - *?$arg_2$: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]*
5. [*?arg_1: (g) $\langle\{ g_1\}, c_g \rangle$ - ATTACK - _: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]* - ATTACK - *_: (g) $\langle\{ g_1\}, c_g \rangle$* - ATTACK - *?$arg_2$: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$]*

.....

As shown in the results, the first and last arguments for each different path are bound to the variables assigned to the respective argument patterns. We use the symbol '_' to denote the intermediate binding arguments. It is obvious that this answer is considerably more detailed about the matching paths, compared to the previous one.

Content rephrasing

ArgQL gives special emphasis to the equivalences. Equivalent propositions are treated as being the same, which means that, a match succeeds not only if a given proposition satisfies the expressed condition, but also if any of its equivalents does. This principle manifests itself in the language in two main ways:

1. In the usage of constant proposition values. Whenever a constant proposition appears in the query that restricts the search for the premises or the conclusion of an argument, that query will succeed if it finds arguments that include the proposition itself, or some equivalent one.

2. In the identification of relations between arguments. That principle arises from the definitions of *attack* and *support* relations and the proposition 2. In particular, regarding the *support* relation, the definition demands equivalence and not equality between the referred propositions. As for the *attack* relation, it is satisfied if there is a conflict between the referred propositions, or according to Proposition 2, between some of their equivalents.

The following query is an example of the first case. In particular, it asks for arguments with conclusion $prop_1$, or some equivalent.

Q_{10}. **match** ?arg: ⟨ ?pr, 'prop_1' ⟩

return ?arg

The answer of this query is the following:

1. [?arg: (a) $\langle\{ a_1, a_2, a_3 \}, c_a \rangle$*]* ,
2. [?arg: (e) $\langle\{ e_1\}, c_a \rangle$*]*

The answer includes the arguments *(a)* and *(e)* because of the equivalence $prop_1 \equiv c_a$ existed in the argument base. Note that the user need not specify this explicitly, but the language will automatically consider all equivalent propositions during the evaluation of the query. The arguments are returned unchanged, exactly as they are stored in the knowledge base. Since there is no argument with conclusion $prop_1$, that value will not appear in the results.

The following query Q_{11}. is an example of the second case. In particular, it searches for arguments that support argument *e*.

Q_{11}. **match** $?arg_1$ *support* $?arg_2$: ⟨ {e_1}, 'c_a' ⟩

return $?arg_1$

for which, the answer is: *[*$?arg_1$*: (f)* $\langle\{ f_1, c_1\}, c_f \rangle$*]*, because of the equivalence $c_f \equiv e_1$

This feature is of great importance in the application domain, because, in spoken human dialogues, even the simplest thing can be expressed in many different ways. This way we bring our query mechanism closer to realistic scenarios and use cases, offering more meaningful searching capabilities.

3.3 Formal specification

In this section we provide a formal description of ArgQL, including both its syntax and semantics. In the first part we define the main constructs of the language and we also give the syntax as an EBNF grammar. In the second part we present the semantics, namely the formal description of how the constructs that we defined in the first part, are evaluated against the data model of section 3.1, in order to locate the correct information. Formal semantics have multiple advantages over documentation written in natural language. They present accurately the behavior of the language, leaving no room for ambiguities, compared to natural language descriptions and also, like in our case, they can be used to study formally possible equivalence with other languages, enabling to prove properties, like soundness and completeness in optimizations etc.

3.3.1 Pattern definitions

ArgQL is based on the idea of pattern matching, an inherently declarative method. Its syntax is mainly influenced by Cypher [48], the language for the Neo4j database and SPARQL [87], the standard query language of the Semantic Web and RDF.

The syntax of ArgQL was designed, having in mind the data model defined in section 3.1. In particular, the most complex and abstract structure in the data model is the argument graph. An *argument graph* is decomposed in *arguments* and *relations*. Arguments are further decomposed in *premises* and *conclusion*, which are both composed by *propositions*. *Relations* are also decomposed in the four different types, which are translated in equivalences or conflicts between some propositions. In ArgQL, we provide a set of patterns-motifs M, each one designed to match with a particular component in the data model. Thus, adopting the same rationale, we have patterns that are composed by simpler ones, and at the same time are part of more complex ones. As a results, M includes patterns that match with simple propositions (*proposition pattern*), with the premise part of arguments *(premise pattern)*, with the conclusion part of arguments *(conclusion pattern)*, with complete arguments *(argument patterns)*, with relations between arguments *(relation pattern)*, with paths *(path pattern)* and with complete dialogues *(dialogue pattern)*. Like any formal language, we also use variables as parts of the different patterns, to bind with data. Let V be the set of ArgQL variables which is infinite. In the following we give formally the definitions for each type of pattern:

Proposition Pattern. A proposition pattern is defined as:

$prp = p$, where $p \in \mathcal{P}$.

It is the simplest pattern type, and matches with proposition p and all of its equivalent ones. We adopt the string representation (”quoted”) for the lexicographic recognition of a proposition pattern in the language.

Argument pattern. Argument patterns are of the most important constructs, since they are used to match with complete arguments in the argument base. An argument pattern is defined as:

$$ap = \begin{cases} v & \text{where } v \in V \text{ a variable} \\ \langle premp, conclp \rangle & \text{where } premp \text{ a premise pattern and } conclp \text{ a conclusion pattern} \end{cases}$$

An argument pattern may be either a single variable v, or be composed by a *premise pattern* and a *conclusion pattern*. In the later case, we use the $\langle\cdot\rangle$ brackets to represent it. The definitions for the *premise* and *conclusion pattern* follow next.

Premise pattern. This type of pattern is used to match with the set of premises of arguments and it is always part of an argument pattern. A premise pattern is defined as:

$$premp = \begin{cases} \{prp_1, \ldots, prp_n\} & \text{where } prp_i \text{ are proposition patterns} \\ v\ [f_{pr}] & \text{where } v \in V \text{ a variable and } [f_{pr}] \text{ a premise filter which is optional} \end{cases}$$

In the first case, the premise pattern is a set of proposition patterns. In this case, the premises of the matching arguments should have propositions with the specific values defined by the prp_i, or some equivalents. In the second case, the premise pattern consists of a variable, followed by a filter. The idea is that variable v will get the whole premise set as value, only if it satisfies the premise filter f_{pr}. The existence of $[f_{pr}]$ is optional. If it does not exist, v matches with the premise set of any argument, without constraints. A *premise filter* is defined as follows:

$$f_{pr} = \begin{cases} f_{incl} = \begin{cases} /\{prp_1, ..prp_n\} & \text{where } prp_i \text{ are proposition patterns} \\ /v' & \text{where } v' \in V \text{ and } v' = premp' \text{ for some } ap' = \langle\ premp', conclp'\ \rangle \end{cases} \\ \\ f_{join} = \begin{cases} .\{prp_1, ..prp_n\} & \text{where } prp_i \text{ are proposition patterns} \\ .v' & \text{where } v' \in V \text{ and } v\text{'} = premp\text{'} \text{ for some } ap' = \langle\ premp', conclp'\ \rangle \end{cases} \end{cases}$$

We support 2 types of premise filters, the *inclusion* (f_{incl}) and the *join* (f_{join}). To discern between them we use the ’/’ symbol for the f_{incl} and the dot (.) symbol for the f_{join}. Both are expressed either with regard to a set of proposition patterns, let s or with a variable v'. In the first case, the filter expresses that s should be included (f_{incl}) or intersect (f_{join}) with the matching premise set of v. In the other case, v' must be the variable that corresponds to the premise pattern of a different argument pattern ap'. For that case, the filter expresses that the matching premise set of v' should be included (f_{incl}) or intersect (f_{join}) with the matching premise set of v.

Conclusion pattern. This type of pattern is used to match with the conclusion of arguments and it is also always part of an argument pattern. A conclusion pattern is defined as:

$$conclp = \begin{cases} prp & \text{where } prp \text{ proposition pattern} \\ v & \text{where } v \in V \text{ a variable} \end{cases}$$

In the first case, the conclusion pattern is a proposition pattern and matches with those arguments for which, the value of their conclusion is the same (or equivalent) to the value of prp. In the second case, the conclusion pattern is a variable and it will match with the conclusion of any argument.

Since we have defined the *premise* and *conclusion patterns*, we now give some examples of complete argument patterns :

Example 3.3.1. · $\langle ?v[/\{"p_1"\}], ?c\rangle$ *: matches arguments whose premises include proposition* $"p_1"$ *or some equivalent.*

· $\langle ?v, "c"\rangle$ *: matches arguments with conclusion any equivalent proposition of "c"*

· $\langle ?v[.\{"p_1", "p_2"\}], ?c\rangle$ *: matches arguments whose premise intersect with a set equivalent to* $\{"p_1", "p_2"\}$

· $\langle \{"p_1", "p_2"\}, "c"\rangle$ *: ground arguments are also considered as argument patterns*

Relation pattern. A relation pattern is used to detect existing relations between arguments and may have one of the following six types:

$$rp = rebut \mid undercut \mid attack \mid endorse \mid back \mid support$$

Note that the *attack* and *support* type express the requirement to match any of the specific sub-relations.

Path pattern. Path patterns are expressions in the language, used to match with complete paths in the graph. A path pattern is defined as follows:

$$pp = \begin{cases} rp & \text{where } rp \text{ relation pattern} \\ pp'/pp'' & \text{where } pp', pp'' \text{ path patterns} \\ pp' * n & \text{where } pp' \text{ path pattern and } n>0 \text{ some integer} \\ pp' + n & \text{where } pp' \text{ path pattern and } n>0 \text{ some integer} \end{cases}$$

In the simplest type, a path pattern is a single relation pattern, requiring for paths with length 1 of the particular relation. Generally a path is characterized as a sequence of relations and we use the delimiter '/' between the relations to denote that sequence. We already described the role of the *n and +n notations, which express the *exactly n* and *at most n* repetitions of the included path pattern. Recall the examples Q_8, Q_9 as examples for their use. The recursive definition of that pattern shows that we can have arbitrary combinations of relation sequences and numerical indicators (e.g. *(attack*2/(support)+2)+3* is an acceptable path pattern). Note that when there is one or more *+n* numerical indicators, it gives a number of different path patterns, equal to the proliferation of the numbers in the '+' indicators. For instance, the previous example defines the union of 6 different path patterns, shown in the following list:

attack/attack/support

attack/attack/support/support

attack/attack/support/attack/attack/support

attack/attack/support/support/attack/attack/support/support

attack/attack/support/attack/attack/support/attack/attack/support

attack/attack/support/support/attack/attack/support/support/attack/attack/support/support

Dialogue pattern The last type of pattern is the dialogue pattern. It is defined as:

$$dp = \begin{cases} ap & \text{where } ap \text{ an argument pattern} \\ pp\ \ dp' & \text{where } pp \text{ a path pattern and } dp' \text{ a dialogue pattern} \end{cases}$$

A *dialogue pattern* can be either a simple argument pattern, or a sequence of alternations between argument patterns and path patterns. We note in this definition that a path pattern always lies between two argument patterns and matches with paths, for which the first and last arguments are bindings for the left and right argument patterns and, additionally, the path between the two matching arguments is the same with one of the paths indicated by *pp*.

The general form of an ArgQL query is given in the following definition:

Definition 21 (ArgQL query)**.** *An ArgQL query q has the form:*

$$\begin{aligned} q \leftarrow\ & \textbf{\textit{match}}\ dp_1,\dots,dp_n \\ & \textbf{\textit{return}}\ x_1,\dots,x_m \end{aligned}$$

where dp_i dialogue patterns and $x_i = v$, with $v \in V$ a variable, or $x_i = path(v',v'')$, with $v',v'' \in V$.

3.3.2 Syntax

In this point we define the concrete syntax of both data and ArgQL in terms of the symbols and keywords used for their lexicographic representation.

Definition 22 (Data syntax)**.** *We denote the syntactic representation of data as a function str, s.t.:*

- *if $e \in P$, with content text, then str(e) = ”text”*
- *if $e \in 2^{\mathcal{P}}$, a set of propositions $e = \{p_1, \dots, p_n\}$, then $str(e) = \{str(p_1), \dots, str(p_n)\}$*
- *if $e \in A$ s.t. $e = (id, pr, c)$, where id is the argument identifier, $pr \in PR$, and $c \in P$, then* $srt(e) = (id)\ \langle str(pr), str(c) \rangle$

For example, the syntactic representation of an argument a is:
$str(a) = (a)\ \langle \{”p_1”, \dots, ”p_n”\}, ”c” \rangle$

Table 3.1 shows the complete syntax of ArgQL in EBNF grammatical rules. The highlighted symbols inside the quotes ’ ’ denote tokens of the language. Anything included in (.)? is optional while the notation (.)* implies zero or more repetitions. The language is not case sensitive, so *”attack”* and *”ATTACK”* are considered the same keyword. Variables are prefixed with ’?’, e.g. *?x*. For constant propositions, we use the quotes ”.” (e.g. ”*p*”).

query	::=	**’MATCH’** (dialoguepattern (’**,**’ dialoguepattern) *
		’RETURN’ returnvalue (’,’ returnvalue)*
dialoguepattern	::=	argpattern |
		argpattern pathpattern dialogue_pattern
argpattern	::=	variable |
		(variable:)? ’**⟨**’ premisepattern ’**,**’ conclusionpattern ’**⟩**’
premisepattern	::=	propset | variable (premisefilter)?
premisefilter	::=	’**[**’ (’**/**’ | ’**.**’) (propset | variable) ’**]**’
conclusionpattern	::=	variable | proposition
propset	::=	**{**’ proposition (’**,**’ proposition)* ’ **}** ’
pathpattern	::=	pp (’**/**’ pp)*
pp	::=	relation |
		’**(** ’pathpattern’**)**’ (’*****’ | ’**+**’) num
returnvalue	::=	variable | ’**PATH**’ ’**(**’ variable ’,’ variable ’**)**’
relation	::=	**’attack’** | **’rebut’** | **’undercut’** |
		’support’ | **’endorse’** | **’back’**
proposition	::=	’”’ .*? ’”’
variable	::=	’**?**’ (’a’ .. ’z’ | ’A’ .. ’Z’ | ’0’ .. ’9’)+

Table 3.1: ArgQL syntax

3.3.3 Semantics

In this section we describe the formal semantics of ArgQL. Before that, we are going to need some new notations for the inclusion and intersection relationships between two sets, such that they will take into account the existing equivalences. In particular, we overload their operators as follows:

$$X \sqsubseteq Y \quad : \quad \text{if } \forall x \in X, \exists y \in Y \text{ s.t. } x \equiv y \quad \text{(Inclusion operator overload)}$$

Intuitively, a set of propositions X is a subset of Y, iff for all propositions in X, there exists an equivalent in Y.
For example, for $X = \{a,b\}$, $Y = \{b,c,d\}$ and $c \equiv a$, it holds that $X \sqsubseteq Y$.

$$X \odot Y \text{ is true} \quad : \quad \text{if} \quad \exists x \in X \text{ and } y \in Y, \text{ s.t. } x \equiv y \quad \text{(Intersection operator overload)}$$

Intuitively, two sets X and Y intersect when there is at least one proposition in X for which there is an equivalent in Y.
For example, for $X = \{a,b,c\}$, $Y = \{d,e\}$ and $e \equiv c$, it holds that $X \odot Y$ is true.

The semantics of ArgQL are based on the interpretation function I_A, which takes as input a pattern $m \in M$ and returns sets of data from A and $G_A$that match m. Moreover, if $S = \mathcal{P} \cup PR \cup A$, s.t. $PR = 2^{\mathcal{P}}$, we also define a variable replacement $\mu : V \mapsto S$, as a function that maps variables to elements of S. The definition of I_A is given in the following list:

- for a *proposition_pattern* $prp = p$ with $p \in \mathcal{P}$ the interpretation is proposition p and all its equivalents as well:

  $$I_A(prp) = \{p' \mid p' \in P(A) \text{ and } p' \equiv p\}$$

  Example 3.3.2. *In the data of the example 3.1.1 the interpretation of the proposition pattern with value 'c_f' is $I_A(t) = \{c_f, e_1, prop_2\}$* □

- for a *premise_pattern premp*:
  - if *premp* is a set of *proposition patterns* $premp = \{prp_1, \ldots, ppr_n\}$:
    $I_A(premp) = \{\{p_1, \ldots p_n\} \mid p_i \in I_A(prp_i) \text{ and } \{p_1, \ldots p_n\} = prem(a) \text{ for some } a \in A\}$

    Intuitively, $I_A(premp)$ returns all sets that match the pattern and constitute the premise of some argument in A.

    Example 3.3.3. *In the data of the example 3.1.1, the interpretation of the premise_pattern $premp = \{'c'_f\}$ is:*
    $I_A(premp) = \{\{e_1\}\}$ since $e_1 \equiv c_f$ and $\{e_1\} = prem(e)$ □
  - if *premp* is of the form $v[f]$, where $v \in V$ and f a *premise_filter*, then
    - for $f = f_{incl}$

- if $premp = v\ [/\{sp_1, \dots sp_m\}]$ a set of proposition patterns, then:

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq props(A),\ \mu(v) = prem(a)$ for some $a \in A$ and $\{s_1, \dots s_m\} \subseteq prem(a)$ with $s_i \in I_A(sp_i)$ and $1 \leq i \leq m\ \}$

  Intuitively, $I_A(premp)$ returns all sets that are premises of some argument in A which also include all propositions that match with the patterns sp_i.

- if $premp = v\ [/v']$, with $v' \in V$ a variable, appearing in the premise pattern of another argument pattern ap':

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq P(A),\ \mu(v) = prem(a)$ for some $a \in A$, $\mu(v') = prem(a')$ for some $a' \in I_A(ap')$ and $\mu(v') \sqsubseteq \mu(v)\}$

  Intuitively, $I_A(premp)$ returns all sets that are premises of some a and include all the premises of another argument a' that matches with the ap'. Note that in the expression $\mu(v') \sqsubseteq \mu(v)$ we used the overloaded subset operator which was defined in the beginning of this section.

∗ for $f = f_{join}$:

- if $premp = v\ [.\{sp_1, \dots sp_m\}]$ a set of proposition patterns, then:

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq P(A),\ \mu(v) = prem(a)$ for some $a \in A$ and $s_i \in prem(a)$, with $s_i \in I_A(sp_i)$, for some $1 \leq i \leq m\ \}$

  Intuitively, $I_A(premp)$ returns all sets that are premises of some argument in A and also include at least one propositions from those matching sp_i.

- if $premp = v\ [.v']$,with $v' \in V$ a variable, appearing in the premise pattern of another argument pattern ap':

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq P(A),\ \mu(v) = prem(a)$ for some $a \in A$, $\mu(v') = prem(a')$ for some $a' \in I_A(ap')$ and $\mu(v') \odot \mu(v)$ is true$\}$

  Intuitively, $I_A(premp)$ returns all sets that are premises of some a and intersect with the premises of another argument a' that matches with the ap_2. Again here, we used the $\odot$ operator which was defined in the beginning of that section to denote the intersection between two sets, taking into account also the equivalences.

Example 3.3.4. *Let two argument patterns $ap_1 : \langle ?pr_1, 'c_c' \rangle$ and $ap_2 : \langle ?pr_2[.?pr_1], ?c \rangle$. In the data of the example 3.1.1 we have that:*

$I_A(?pr_2[.?pr_1]) = \{\{'f_1', 'c_1'\}\}$

In particular, $\mu(?pr_1) = \{c_1, c_2\} = prem(c)$, since $c \in ap_1$ and $\mu(?pr_2) = \{f_1, c_1\} = prem(f)$ and it holds that $prem(c) \odot prem(f)$ is true. □

- for a *conclusion_pattern conclp*:

- if *conclp* is a *proposition_pattern prp*, then:

  $$I_A(conclp) = \{p \in I_A(prp) \mid concl(a) = p \text{ for some } a \in A\}$$

- if t is a *variable* v, then:

  $$I_A(conclp) = \{\mu(v) \in P(A) \mid \mu(v) = concl(a), \text{ for some } a \in A\}$$

- for an *argpattern ap*:
  - if *ap* is a single *variable* $v \in V$ then: $I_A(ap) = A$
  - if *ap* is of the form $\langle premp, conclp\rangle$, where *premp* a *premise_pattern* and *conclp* a *conclusion_pattern*, then:
    $I_A(ap) = \{a \in A \mid prem(a) \in I_A(premp) \text{ and } concl(a) \in I_A(conclp)\}$

Example 3.3.5. *Recalling the example 3.3.4 for the data in the example 3.1.1, we have that* $I_A(ap_1) = \{c\}$ *and* $I_A(ap_2) = \{f\}$. □

- for a *relation* pattern:
  - $I_A(rebut) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A \text{ s.t. concl}(a_1) \not\sim \text{concl}(a_2)\ \}$
  - $I_A(undercut) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A \text{ s.t. concl}(a_1) \not\sim p_i \text{ for some } p_i \in \text{prem}(a_2)\}$
  - $I_A(attack) = I_A(rebut) \cup I_A(undercut)$
  - $I_A(endorse) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A \text{ s.t. concl}(a_1) \equiv \text{concl}(a_2)\ \}$
  - $I_A(back) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A \text{ s.t. concl}(a_1) \equiv p_i \text{ for some } p_i \in \text{prem}(a_2)\ \}$
  - $I_A(support) = I_A(endorse) \cup I_A(back)$
- for a *pathpattern pp*.
  - if *pp* is a *relation* pattern of type *rp*, then $I_A(pp) = I_A(rp)$
  - if *pp* is of the form pp_1/pp_2, for pp_1, pp_2 *pathpatterns* then

    $$I_A(pp_1/pp_2) = \{P_{a_1 \to a_n} = P_{a_1 \to a_m} \cdot P_{a_m \to a_n} \mid P_{a_1 \to a_m} \in I_A(pp_1) \text{ and } P_{a_m \to a_n} \in I_A(pp_2)\}$$

  - if *pp* is of the form $pp * n$, for *pp* path pattern, which is translated into $pp/\ldots/pp$ (n times) we have:
    - if $n = 1$, then $I_A(pp * 1) = I_A(pp)$
    - if $n > 1$, then $I_A(pp * n) = I_A(pp/pp * (n-1))$
  - if *pp* is of the form $pp + n$, which is translated into a set of pathpatterns $\bigcup_{k=1}^{n}(pp * k)$:
    $I_A((pp) + n) = \bigcup_{k=1}^{n} I_A(pp * k)$
- for a *dialogue_pattern dp*, we have that:

  With *head(dp)* we denote the first argument pattern that appears in the sequence of argument patterns in the dialogue pattern dp. We have:

- if dp an argument pattern ap, then: $I_A(t) = \{P_{a \rightarrow a} = a \mid a \in I_A(ap)\}$
  (based on the definition 19 for path)

- if dp is $ap\ pp\ dp'$, then:
  $I_A(dp) = \{P_{a_0 \rightarrow a_n} = (a_0\ a_1\ \ldots\ a_x \ldots\ a_n) \mid a_0 \in I_A(ap), P_{a_0 \rightarrow a_x} \in I_A(pp), a_x \in I_A(head(dp'))$ and $P_{a_x \rightarrow a_n} \in I_A(dp')\}$

Example 3.3.6. *Let a dialogue pattern*
`dp:  ?a`$_1$` attack/attack ?a`$_2$`:⟨?pr`$_2$`, 'c`$_a$`'⟩ attack ?a`$_3$

dp is decomposed into the following patterns: The argpattern `ap`$_1$ = `?a`$_1$*, the path pattern* `pp`$_1$ = `attack/attack` *and a second dialogue pattern* `dp`$_2$ = `?a`$_2$`:⟨?pr`$_2$`, 'c`$_a$`'⟩ attack ?a`$_3$*, for which* $head(dp_2)$ =*?*a_2*. We have that:*

- $I_A(dp_2) = \{$*(a b), (a g)* $\}$
- $I_A(head(dp_2)) = \{a\}$
- $I_A(ap_1) = \{a,b,c,d,e,f,g\}$
- $I_A(pp_1) = \{$ *(a b a), (a g a), (b a g), (g a b), (c b a), (d c d), (d c b)* $\}$

Combining them altogether, we have that $I_A(dp) = \{$ *(a rebut b rebut a rebut b), (a rebut b rebut a rebut g), (a rebut g rebut a rebut b), (a rebut g rebut a rebut g), (c undercut b rebut a rebut b), (c undercut b rebut a rebut g)*$\}$ □

Recalling the definition of the match clause of an ArgQL query:

match $dp_1, \ldots dp_n$

where dp_i *dialogue_patterns*, we define the semantics of *match*, as:

$$I_A(match) = \{d_1, \ldots, d_n \mid d_1 \in I_A(dp_1), \ldots d_n \in I_A(dp_n)\}$$

3.3.4 Results Form

Let the set $var(dp) \subseteq V$, be the set of variables appearing in a dialogue pattern dp. The *return* statement of an ArgQL query q has the form:

$$\textbf{\textit{return}}\ x_1, \ldots, x_m$$

where $x_i = v$, with $v \in V$ a variable, or $x_i = path(v, v')$, with $v, v' \in V$.
We denote with $r \in I_A(match)$ a row of the matching data and with $\pi_r(x_i)$ the projection of the element x_i in r. We also recall the syntactic representation function for data *str* of the definition 22. It holds that:

- if x_i a variable v that corresponds to an argument pattern *ap*, then $\pi_r(x_1) = v : str(a)$, where a is the value of x_i in r, for which $a \in I_A(ap)$
- if x_i a variable v that corresponds to a premise pattern *premp*, of an argument pattern ap appearing in the query, then $\pi_r(x_i) = v : str(pr)$, where $pr = prem(a)$ for a the argument value of ap in r, for which $a \in I_A(ap)$
- if x_i a variable v that corresponds to the conclusion pattern *conclp*, of an argument pattern ap appearing in the query, then $\pi_r(x_i) = v : str(c)$, where $c = concl(a)$ for a the argument value of ap in r, for which $a \in I_A(ap)$
- if x_i has the form *path*(v_1, v_2), which means that some of the dp_j is a path pattern *pp* which lies between two argument patterns indicated by the variables v_1, v_2. Let $p \in I_A(pp)$ the path value of *pp* that appears in r. Assuming that the satisfied path pattern is of the form $r_1/r_2/\ldots/r_k$, where r_1 one of the *rebut, undercut, attack, endorse, back, support* and that $p = (a_1\ a_2\ \ldots\ a_k\ a_{k+1})$, we have that:

  $$\pi_r(x_i) = v_1 :\ (a_1)str(a_1)\quad r_1\quad \_ :\ (a_2)str(a_2)\quad \ldots\quad \_ :\ (a_k)str(a_k)\quad r_k\quad v_2 :\ (a_{k+1})str(a_{k+1})$$

  Note that the $\_ :$ notations refer to the intermediate argument patterns generated during translation.

Finally, we define the answer to the query q as following:

$$ans(q) \leftarrow \{\ [\ \pi_r(x_1), \ldots, \pi_r(x_n)\ ]\ |\ r \in I_A(match)\}$$

Chapter 4
Theoretic principles for Query Execution

Contents

In the previous chapter, we gave the formal specification of ArgQL. In the next stage, we deal with issues related to how ArgQL queries will be actually executed. For several reasons that were explained in the introduction, we chose, instead of building everything from scratch, to pursue a transition to some of the standard storage models and take advantage of their well-known capabilities and efficiency. In this work, we show the case of the RDF storage scheme. Since ArgQL is implementation-agnostic and representation-agnostic, i.e., independent from the specific model used to represent data, the translation process can be easily regenerated to a different storage scheme (e.g. relational), based on the methodology that we propose here. This chapter is devoted to present this translation, while in the next one, we describe how this methodology is incorporated in the implementation of ArgQL. In particular, in section 4.1 we give all the formal details of the process, along with examples to enforce the comprehensibility. In the second part of this chapter (section 4.2), we propose an optimization on top of the proposed translation. The optimization aims at generating shorter queries than those generated by the initial approach, which by extension, reflects as an improvement in the efficiency of the execution. That improvement is verified at the experimental evaluation, which is described in section 5.2.

4.1 Mappings to the RDF storage scheme

In this section we will define the theoretical ground that supports our ArgQL-to-SPARQL translation. To do this, we break the process in three main sub-tasks that need to be addressed. At first, in section 4.1.1 we define the mapping between our data model of section 3.1 and RDF. We adopt an ontological scheme to represent concepts in the domain and we show the mappings between the different elements in our data model and the concepts of this scheme. Next, in section 4.1.2, we define the translation between the query languages. The translation is presented as a set of rules that describe each different pattern type of the ArgQL specification, to which graph pattern it corresponds, which targets the ontological scheme we defined in the previous stage. Finally, in .2, we prove the properties of soundness and completeness of the proposed methodology. In order to prove these properties, we use the data and query translations, as well as the formal semantics of ArgQL and SPARQL.

4.1.1 Data mapping

We now show the mapping between our data model and RDF. In order to express data in RDF, we need to define a scheme with the basic concepts of the referred domain. To this end, we use the conceptualization of AIF, described in section 2.1.4. We ended up in this choice for three main reasons:

- AIF has become a standard language in the area of argumentation for representing arguments and dialogues. The goal of AIF, is to provide an interlingua for the different argument representations, that will enhance the mapping among the various theoretic argumentation frameworks, as well as the interoperability among the multitude of argumentation tools existing in the area. A mapping from our model to the AIF/RDF representation, would naturally make ArgQL compliant with all those models and by extension the community of the Argument Web [22], increasing that way its usability.
- Although the concepts of AIF and its extension AIF+ (sections 2.1.4 and 2.1.4) are relatively simple in their representation, they carry quite rich semantics, that are able to represent the greatest portion of the proposed argumentation models and as a result, there is a significant correspondence with the concepts of our data model, as well.
- We take advantage of the AIFdb Corpora[1], a quite large corpora that includes argumentative data, gathered by a number of argumentation tools, which are structured in AIF format [67], allowing us to test an implementation of ArgQL on existing data sets.

In this part, we type the classes and the relations of AIF, according to the names used in AIFdb Corpora. We recall $V_{AIF+} = I \cup RA \cup CA \cup PA \cup L \cup YA \cup TA \cup MA$ the set of the various class types in the AIF+ model. The mapping is defined at the level of the language $\mathcal{L} =< \mathcal{P}, Cn >$ and it is presented

[1]http://corpora.aifdb.org/

in table 4.1. In the table we define a set of *mapping rules*, that associate the elements of $\mathcal{L}$ with the respective concepts in the AIF conceptualization. In particular, the first column *(Rule)* includes the name of the mapping rule. The second column *(Element in $\mathcal{L}$)* includes the particular element x in $\mathcal{L}$ that is being mapped. The third column *(AIF graph)* is a semi-formal description of the graph to which x is mapped, which is essentially an AIF-graph of the definition 6 of section 2.1.4. With the notation $\hat{x}$ in the third column, we refer to the instance from the AIF classes, to which x is mapped, namely $\hat{x} \in V_{AIF+}$. Furthermore, the part after the : describes any additional instances and relations between them, that represent the AIF-graph to which x is mapped. As a result, any $x \xrightarrow{t} x$ denotes an edge of type t between the nodes $x, y \in V_{AIF+}$. Finally, the last column includes the representation of that AIF-graph in RDF. We denote as $\|x\|$ the RDF representation of the AIF graph described in the third column. By the definition 7 of section **??** according to which an RDF graph G consists of a set of triples, we have that $\|x\| \subseteq G$. Recalling also the set U of the RDF's identifiers, URIs, we use the function $u : V_{AIF+} \to U$ to assign RDF identifiers to the resources generated during the mapping process. Figure 4.1 illustrates the AIF-graphs generated by the mapping.

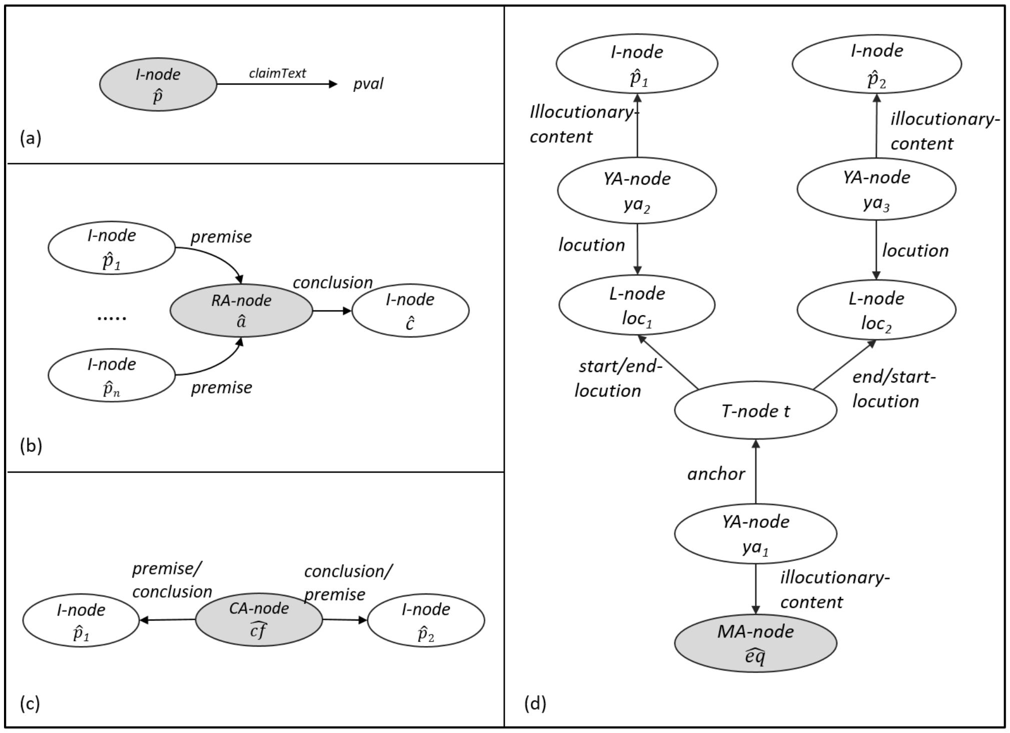


Figure 4.1: AIF graphs

r_1 states that a proposition $p \in \mathcal{P}$ is mapped to an I-node instance, denoted as $\hat{p} \in I$, for which the *claimText* value is p. The rule r_2 states that an argument a in $\mathcal{L}$ is associated to an instance of the *RA-node* class, namely it holds that $\hat{a} \in RA$. For the RA-node instance $\hat{a}$, there are n incoming edges of type *premise* from the I-nodes $\hat{p}_1, \ldots, \hat{p}_n \in I$ and an outgoing edge of type *conclusion* to

Rule	Element in $\mathcal{L}$	AIF graph	RDF representation
(r_1)	$p \in \mathcal{P}$	$\hat{p} \in I$: $\hat{p} \xrightarrow{\text{claimText}} p$ (fig. 4.1a)	$\lVert p \rVert$ = $(u(\hat{p})$ *type I-node*), $(u(\hat{p})$ *claimText* $p)$
(r_2)	$a = \langle\{p_1,\dots,p_n\},c\rangle$ s.t. $c \in Cn(\{p_1,\dots,p_n\})$	$\hat{a} \in RA$: $\hat{p}_1 \xrightarrow{\text{premise}} \hat{a}$, … , $\hat{p}_n \xrightarrow{\text{premise}} \hat{a}$, $\hat{a} \xrightarrow{\text{conclusion}} \hat{c}$ (fig. 4.1b)	$\lVert a \rVert$ = $(u(\hat{p}_1)$ *premise* $u(\hat{a}))$ … $(u(\hat{p}_n)$ *premise* $u(\hat{a}))$ $(u(\hat{c})$ *conclusion* $u(\hat{a}))$, $(u(\hat{a})$ *type RA-node*)
(r_3)	$cf = (p_1,p_2)$ s.t. $p_1 \nleftarrow p_2$	$\hat{cf} \in CA$: $\hat{cf} \xrightarrow[\text{conclusion}]{\text{premise}} \hat{p}_1$, $\hat{cf} \xrightarrow[\text{premise}]{\text{conclusion}} \hat{p}_2$ (fig. 4.1c)	$\lVert cf \rVert$ = $(u(\hat{cf})$ *premise* $u(\hat{p}_1))$, $(u(\hat{cf})$ *premise* $u(\hat{p}_2))$, $(u(\hat{cf})$ *conclusion* $u(\hat{p}_1))$, $(u(\hat{cf})$ *conclusion* $u(\hat{p}_2))$, $(u(\hat{cf})$ *type CA-node*)
(r_4)	$e = (p_1,p_2)$ s.t. $p_1 \equiv p_2$	$\hat{e} \in MA$: $ya_1 \xrightarrow{\text{illocContent}} \hat{p}_1$, $ya_2 \xrightarrow{\text{illocContent}} \hat{p}_2$ $ya_1 \xrightarrow{\text{locution}} loc_1$, $ya_2 \xrightarrow{\text{locution}} loc_2$, $ta \xrightarrow[\text{endLocution}]{\text{startLocution}} loc_1$, $ta \xrightarrow[\text{startLocution}]{\text{endLocution}} loc_2$ $ta \xrightarrow{\text{anchor}} ya_3$, $ya_3 \xrightarrow{\text{illocContent}} \hat{e}$, and $ya_1,ya_2,ya_3 \in YA$, $loc_1,loc_2 \in L$ and $t \in T$ (fig. 4.1d)	$\lVert e \rVert$ = $(u(ya_1)$ *illocContent* $u(\hat{p}_1))$, $(u(ya_2)$ *illocContent* $u(\hat{p}_2))$, $(u(ya_1)$ *type YA-node*), $(u(ya_2)$ *type YA-node*), $(u(ya_1)$ *locution* $u(loc_1))$, $(u(ya_2)$ *locution* $u(loc_2))$, $(u(loc_1)$ *type L-node*), $(u(loc_2)$ *type L-node*), $(u(ta)$ *type TA-node*), $(u(ta)$ *startLocution* $u(loc_1)$), $(u(ta)$ *endLocution* $u(loc_1)$), $(u(ta)$ *startLocution* $u(loc_2)$), $(u(ta)$ *endLocution* $u(loc_2)$), $(u(ya_3)$ *type YA-node*), $(u(ta)$ *anchor* $u(ya_3))$, $(u(ya_3)$ *illocContent* $u(\hat{e}))$, $(u(\hat{e})$ *type MA-node*)

Table 4.1: Mapping rules

the respective I-node $\hat{c} \in I$. The rule r_3 says that a conflict c between two propositions p_1 and p_2, is mapped to an AIF conflict represented as a CA-node instance. That means $\hat{c} \in CA$, and the I-nodes $\hat{p}_1, \hat{p}_2 \in I$ connect with $\hat{c}$ via *premise* and *conclusion* edges. Note that the property of symmetry of the conflict in our data mode, is preserved in the generated graph. Similar rationale holds also for rule

r_4, which represents the translation for the equivalence between two propositions p_1, p_2. In this case, new instances are created of the classes *YA, L* and *T*, which are required to represent the notion of equivalence (case of *default rephrase* in AIF+ of section 2.1.4).

Given all the above, we have that an argument base A of the definition 17 is transformed to an RDF graph, according to the following definition.

Definition 23 (Argument Base to RDF graph)**.** *Given an argument base A of a finite set of arguments and the mapping $\|\cdot\|$ to an RDF graph, we have that:*

$$\|A\| = \bigcup_{a\in A}\|a\| \cup \bigcup_{c\in cf(A)}\|c\| \cup \bigcup_{e\in eq(A)}\|e\|$$

where cf(A) and eq(A) finite sets of conflicts and equivalences between the propositions in the arguments of A, respectively.

Notations: With $df(i_1, val_1, i_2, val_2)$ we will refer to the AIF's *default rephrase*, between i_1, i_2, with *claimText* values val_1, val_2, respectively. With $ca(i_1, i_2)$, we will refer to the *conflict application* between the two I-nodes i_1, i_2.

Next, we make some observations about the mapping as defined above and discuss them in two main axes: *information preservation* and *mapping reversibility*.

Information preservation

A common risk when transforming data to a different format is the possibility of information loss. One of the highest priorities in such a process is the assurance of data integrity, which means that, apart from reproducing the content in its entirety, it also crucial to retain any preexisting properties. The proposed mapping satisfies this requirement and we justify this observation for each particular case of table 4.1.

The case of propositions is obvious. Each $p \in \mathcal{P}$ produces an I-node. The content of p is transferred in the *claimText* property of i and there are no particular properties related to propositions that should be transferred to the AIF representation. Concerning the second case, any argument a generated in $\mathcal{L}$ has an equivalent *RA-node*, which has a number of incoming edges from I-nodes that corresponds to the propositions in $prem(a)$ and an outgoing edge to an I-node that correspond the proposition $concl(a)$. There is also an analogy between the semantics of *Cn* and the *RA-node* class, since they both express the general notion inference, without restricting it to some of the *modus ponens, defeasible etc.*. Finally, any relation of conflict and equivalence has its own representation in AIF model. It is also noticeable that the property of symmetry is preserved between the I-nodes that correspond to the involved propositions. This is achieved by using the same types of edges for both directions (the edges *premise/conclusion* for the conflict and the edges *start/end-locution* for the equivalence).

Mapping reversibility

An interesting question considering the proposed mapping, is whether it can also be applied backwards, which means, if for any AIF graph described in the right part of a rule (columns 3, 4), there is an equivalent element of the left part in $\mathcal{L}$ (column 2). Again here, the answer to this question depends on whether there is semantic correspondence between the two models.

Regarding the rule r_1, any I-node with a particular value in its *claimText* property, can be associated to a proposition in $\mathcal{P}$ with the same value. The same holds for r_2. In particular, any argument expressed as an instance of RA can also be an argument of $\mathcal{L}$ since, as we mentioned above, both *Cn* nor *RA-node* generalize the inference relation and any particular rule of inference, like defeasible, deductive, modus ponens etc. constitutes special case. On the other hand, we are not able to answer about the reversibility of r_3 and r_4. This is because, unlike in $\mathcal{L}$, the semantics of *CA* and *default rephrase* have not been formally defined. Thus we can not deduce that for any expressed conflict or default rephrase in AIF, the conditions in definitions 12 and 13 pf section 3.1 are satisfied.

4.1.2 ArgQL to SPARQL Translation

After we showed the mapping between the argumentation data model described in section 3.1 and the RDF scheme, we now present in a formal way the translation between the associated query languages, namely ArgQL and SPARQL. The translation is shown in a declarative way. The general idea is that each different pattern of ArgQL is translated to a particular graph pattern of SPARQL and, as the different ArgQL patterns are combined from the simpler to the most general ones, the respective graph patterns are also combined and they progressively build the final SPARQL query. Key challenges throughout the process is to define properly these graphs patterns, as well as to make the appropriate joins, to ensure the correctness of the results. In the appendix .2 we prove that the proposed translation indeed succeeds the expected results.

In order to define the process formally, we adopt some notations. The basic symbol which represents the translation between the two languages is the $\langle\!\langle\cdot\rangle\!\rangle$. In particular, recalling the set M of the available ArgQL patterns and assuming $m \in M$ we denote as $\langle\!\langle m\rangle\!\rangle$ the SPARQL graph pattern, to which m corresponds. One of the most crucial parts in the process is the way that variables are being used. Variables that appear in the ArgQL query will remain also in SPARQL, but will be put in those positions such that they will get the respective values. Beyond that, new variables will have to be created in the SPARQL query, which, in most of the cases, are used for the URI identifiers, and which are necessary in order to make the required joins. For the sake of simplicity and disambiguation, we assume that the sets V and W that correspond to the ArgQL and SPARQL variables respectively, are disjoint. For the SPARQL variables, we will use the representation $w_t \in W$. Subscript t will be used to indicate the resource type identified by the matching URIs and can be one of *(i, ra, ca, ma, pa)*, indicating the URIs of an I-node instance, an RA-node instance, a CA-node instance, an MA-node instance and a PA-node instance, respectively. By the notation *gp.w* we access the variable w of the graph pattern *gp*.

In order to express in the rules that two graph patterns join on some variable, we adopt the following convention. Let gp be a graph pattern, that includes variable w and also uses the graph pattern gp' in its definition. Assume also that gp' includes variable w'. In order to express that gp and gp' join on variables w and w', we write is as a subscript on gp', in which we denote the equality between the variables. In the particular we write:

$$gp = \ldots w \ldots gp'_{w'=w} \cdots$$

A convention here is that the first (left) variable in the equality of the subscript (w'), must always be the one defined in the graph pattern to which the subscript it attached gp'. Furthermore, if we want to express that they join on more than one variables, we will write it as $gp'_{w'_1=w_1;w'_2=w_2;\ldots}$, again with the left variables in all of the equalities having been defined in gp'. Finally, we use the symbol $\wedge$, to denote the *conjunction* between graph patterns (the *AND* in SPARQL terminology) and the $\vee$ to denote their disjunction (*UNION*).

The complete translation of ArgQL into SPARQL is presented in tables 4.2, 4.3, 4.4 and 4.5. In particular, these tables include a set of transformation rules that define the translation of the various ArgQL pattern types. Afterwards, we explain it informally along with some examples to enhance the comprehensibility of the process.

The first two rules (r_5, r_6) have an auxiliary role in the translation process. In particular, the requirement to match with some equivalence or conflict is quite often in ArgQL and that requirement is transferred to SPARQL, as well. For simplicity, in order to avoid repeating their translation code in multiple rules, we record them as separate rules, and we will use them through their notation, anywhere needed. Therefore, only for the translation purposes, we assume two additional patterns, *conflict pattern* and *equivalence pattern* and their translation is given in those two rules. About r_5, the translation creates a graph pattern that will match *any* existing conflict between the two I-nodes matching with w_{i1} and w_{i2}. In addition, the translation for r_6 will match any *default rephrase* (as the notion of equivalence in the AIF specification is called) between the I-nodes matching with w_{i1} and w_{i2} of the respective graph pattern.

Rule r_7 is the translation of a proposition pattern with of a proposition $p \in \mathcal{P}$ and the resulting graph pattern expresses the matching to any I-node with the value p in the *claimText* property. We also define the $\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle$ for the graph pattern that retrieves the *default rephrase* of the I-node to which the $\langle\!\langle prp \rangle\!\rangle$ is expected to match.

Rule r_8 defines the translation of a *premise pattern*. We discriminate the two different forms of a premise pattern, namely the case of the set of proposition patterns and the form $v_p[f_{pr}]$, because they are translated to different graph patterns. About the ($r_{8.1}$) we have that, for each of the I-nodes corresponding to the proposition patterns prp_j, we must also retrieve the equivalents to check whether they are premises of the particular ra. For this reason, we also include the $\langle\!\langle eqp \rangle\!\rangle$ of the rule r_6 indicating the one of the two instances (i_1) must be the one in prp_j, and the other (i_2), must be the equivalent and check which of these (union) is a premise of w_{ra}. In addition, the *not exists* part of the translation describes that there must not be some other I-node which will be different from these in the given set, that is also a premise of w_{ra}, because the filter requires from the matching premises to

(r_5) Conflict pattern ***conflp***

$\langle\!\langle \textbf{conflp} \rangle\!\rangle$ = $(w_{ca}$ *type CA-node*$) \wedge (w_{ca}$ *premise* $w_{i1}) \wedge (w_{ca}$ *conclusion* $w_{i1}) \wedge (w_{ca}$ *premise* $w_{i2})$ $\wedge\ (w_{ca}$ *conclusion* $w_{i2})$

(r_6) Equivalence pattern ***eqp***

$\langle\!\langle \textbf{eqp} \rangle\!\rangle$ = $(w_{ya1}$ *illocutionaryContent* $w_{i1}) \wedge (w_{ya2}$ *illocutionaryContent* $w_{i2})$ $\wedge\ (w_{ya1}$ *type YA-node*$) \wedge (w_{ya2}$ *type YA-node*$) \wedge (w_{ya1}$ *locution* $w_{loc1}) \wedge (w_{ya2}$ *locution* $w_{loc2}) \wedge$ $(w_{loc1}$ *type L-node*$) \wedge (w_{loc2}$ *type L-node*$) \wedge (w_{ta}$ *type TA-node*$) \wedge (w_{ta}$ *startLocution* $w_{loc1}) \wedge$ $(w_{ta}$ *endLocution* $w_{loc1}) \wedge (w_{ta}$ *startLocution* $w_{loc2}) \wedge (w_{ta}$ *endLocution* $w_{loc2}) \wedge$ $(w_{ya2}$ *illocutionaryContent* $w_{i2}) \wedge (vya3$ *type YA-node*$) \wedge (w_{ta}$ *anchor* $w_{ya3}\ ) \wedge$ $(w_{ya3}$ *illocutionaryContent* $w_{ma}) \wedge (w_{ma}$ *type MA-node*$)$

(r_7) Proposition pattern ***prp*** **=** p with $p \in \mathcal{P}$

$\langle\!\langle \textbf{prp} \rangle\!\rangle$ = $(w_i$ *type I-node*$) \wedge (w_i$ *claimText* $p)$

$\langle\!\langle \textbf{prp} \rangle\!\rangle.\langle\!\langle \textbf{eqp} \rangle\!\rangle = \langle\!\langle prp \rangle\!\rangle \wedge \langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle prp \rangle\!\rangle.w_i}$

(r_8) Premise pattern ***premp*** of an argument pattern ap

$(r_{8.1})$ *premp* is a set $\{prp_1,\ldots,prp_n\}$, where prp_j proposition patterns of rule r_7

$$\langle\!\langle \textbf{premp} \rangle\!\rangle = \Big[\bigwedge_{j=1}^{n} \big(\ \langle\!\langle prp_j \rangle\!\rangle \wedge \langle\!\langle prp_j \rangle\!\rangle.w_i \textit{ premise } w_{ra}\big)\ \big) \vee$$

$$\big(\ \langle\!\langle prp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle \wedge (\ \langle\!\langle prp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle.w_{i2} \textit{ premise } w_{ra}) \wedge \big)\Big] \wedge$$

$$\textit{NOT EXISTS}\ \Big((\bigwedge_{j=1}^{n} w_x \neq \langle\!\langle prp_j \rangle\!\rangle.w_i) \ \wedge\ (w_x \textit{ premise } w_{ra})\ \Big) \wedge (w_{ra} \textit{ type RA-node})$$

$(r_{8.2})$ *premp* is of the form $v_p[f_{pr}]$, where $v_p \in V$ and f_{pr} a premise filter

$$\langle\!\langle \textbf{premp} \rangle\!\rangle = (w_i \textit{ type I-node}) \ \wedge\ (w_i \textit{ claimText } v_p) \ \wedge\ (w_i \textit{ premise } w_{ra}) \ \wedge$$
$$(w_{ra} \textit{ type RA-node}) \ \wedge\ \langle\!\langle f_{pr} \rangle\!\rangle_{w_{ra}}$$

(r_9) Premise filter $\boldsymbol{f_{pr}}$ coming from the premise_pattern ***premp*** $= v_p[f_{pr}]$, with $v_p \in V$ such that:

f_{pr} = ('/' | '.') *(proposition_set | variable)*

$(r_{9.1})$ for $f_{pr} = [/\{sp_1,\ldots,sp_m\}]$ an *inclusion* filter based on a set of proposition patterns sp_j.

$$\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle = \bigwedge_{j=1}^{m} \Big[\ \big(\ \langle\!\langle sp_j \rangle\!\rangle \wedge (\langle\!\langle sp_j \rangle\!\rangle.w_i \textit{ premise } w_{ra})\ \big)\ \vee$$

$$\big(\ \langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle \wedge (\ \langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle.w_{i2} \textit{ premise } w_{ra}\ )\ \big)\Big]$$

Table 4.2: ArgQL to SPARQL Translation

(r_9) Premise filter $\boldsymbol{f_{pr}}$ coming from the premise_pattern ***premp*** $= v_p[f_{pr}]$, with $v_p \in V$ such that:

$$f_{pr} = (\text{'/'} \mid \text{'.'})\ (\mathit{proposition\_set} \mid \mathit{variable})$$

$(r_{9.2})$ for $f_{pr} = [/v_2]$, an *inclusion* filter based on a different premise pattern $\mathit{premp}_2 = v_2[f_{pr2}]$ of the rule $(r_7.2)$

$$\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle = \mathit{NOT\ EXISTS}\ \{\ (w_i\ \mathit{premise}\ \langle\!\langle \mathit{premp}_2 \rangle\!\rangle.w_{ra}) \wedge \\ \mathit{NOT\ EXISTS}\ \{\ (w_i\ \mathit{premise}\ w_{ra}) \vee \\ \Big(\ \langle\!\langle \mathit{eqp} \rangle\!\rangle_{w_{i1}=w_i} \wedge (\langle\!\langle \mathit{eqp} \rangle\!\rangle.w_{i2}\ \mathit{premise}\ w_{ra})\ \Big)\ \}\ \}$$

$(r_{9.3})$ for $f_{pr} = [.\{sp_1,\ldots,sp_m\}]$ a *join* filter based on a set of proposition patterns sp_j of the rule r_7.

$$\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle = \bigvee_{j=1}^{m} \Big[\ \Big((\langle\!\langle sp_j \rangle\!\rangle \wedge (\langle\!\langle sp_j \rangle\!\rangle.w_i\ \mathit{premise}\ w_{ra})\ \Big) \vee \\ \Big(\ \langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle \mathit{eqp} \rangle\!\rangle \wedge (\ \langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle \mathit{eqp} \rangle\!\rangle.w_{i2}\ \mathit{premise}\ w_{ra}\ )\ )\ \Big)\Big]$$

$(r_{9.4})$ for $f_{pr} = [.v_{p2}]$, a *join* filter based on a different premise pattern $\mathit{premp}_2 = v_2[f_{pr2}]$ of the rule $(r_7.2)$

$$\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle = (\langle\!\langle \mathit{premp}_2 \rangle\!\rangle.w_i\ \mathit{premise}\ w_{ra}) \vee \\ \Big(\ \langle\!\langle \mathit{eqp} \rangle\!\rangle_{w_{i1}=\langle\!\langle \mathit{premp}_2 \rangle\!\rangle.w_i} \wedge (\langle\!\langle \mathit{eqp} \rangle\!\rangle.w_{i2}\ \mathit{premise}\ w_{ra})\ \Big)$$

(r_{10}) Conclusion pattern ***conclp***

$(r_{10.1})$ for *conclp* a *proposition* pattern of r_6, let *prp*

$$\langle\!\langle \boldsymbol{\mathit{conclp}} \rangle\!\rangle = \Big(\langle\!\langle \mathit{prp} \rangle\!\rangle \wedge (w_{ra}\ \mathit{conclusion}\ \langle\!\langle \mathit{prp} \rangle\!\rangle.w_i)\ \Big) \vee \\ \Big(\ \langle\!\langle \mathit{prp} \rangle\!\rangle.\langle\!\langle \mathit{eqp} \rangle\!\rangle \wedge (w_{ra}\ \mathit{conclusion}\ \langle\!\langle \mathit{prp} \rangle\!\rangle.\langle\!\langle \mathit{eqp} \rangle\!\rangle.w_{i2}) \wedge (w_{ra}\ \mathit{type\ RA\text{-}node})\ \Big)$$

$(r_{10.2})$ for *conclp* a variable $v_c \in V$, then:

$$\langle\!\langle \boldsymbol{\mathit{conclp}} \rangle\!\rangle = (w_i\ \mathit{type\ I\text{-}node}\ ) \wedge (w_i\ \mathit{claimText}\ v_c) \wedge (w_{ra}\ \mathit{conclusion}\ w_i) \wedge \\ (w_{ra}\ \mathit{type\ RA\text{-}node})$$

(r_{11}) Argument pattern $ap =< premp, conclp >$, with *premp* premise pattern of r_8 and *conclp* a conclusion pattern of r_9.

$$\langle\!\langle \boldsymbol{ap} \rangle\!\rangle = \langle\!\langle \mathit{premp} \rangle\!\rangle \wedge \langle\!\langle \mathit{conclp} \rangle\!\rangle_{w_{ra}=\langle\!\langle \mathit{premp} \rangle\!\rangle.w_{ra}}$$

(r_{12}) Relation pattern ***rp*** between two **arbitrary** argument patterns $ap_1 =< premp_1, cp_1 >$ and $ap_2 =< premp_2, cp_2 >$, with $premp_1$, $premp_2$ premise patterns and cp_1, cp_2 conclusion patterns.

$(r_{12.1})$ for *rp* a *rebut* relation

Table 4.3: ArgQL to SPARQL Translation

(r_{12}) Relation pattern ***rp*** between two **random** argument patterns $ap_1 =< premp_1, cp_1 >$ and $ap_2 =< premp_2, cp_2 >$, with $premp_1$, $premp_2$ premise patterns and cp_1, cp_2 conclusion patterns.

$(r_{12.1})$ for *rp* a *rebut* relation

$$\langle\!\langle \textbf{\textit{rebut}} \rangle\!\rangle = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \Big[\Big(\langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i \;;\; w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \Big) \vee$$
$$\Big(\langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i \;;\; w_{i2}=\langle\!\langle eqp \rangle\!\rangle.w_{i2}} \Big) \vee$$
$$\Big(\langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle eqp \rangle\!\rangle.w_{i2} \;;\; w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \Big) \vee$$
$$\Big(\langle\!\langle eqp_1 \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge \langle\!\langle eqp_2 \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle eqp_1 \rangle\!\rangle.w_{i2} \;;\; w_{i2}=\langle\!\langle eqp_2 \rangle\!\rangle.w_{i2}} \Big) \Big]$$

$(r_{12.2})$ for *rp* an *undercut* relation.

$$\langle\!\langle \textbf{\textit{undercut}} \rangle\!\rangle = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \Big[\Big(\langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i,\, w_{i2}=\langle\!\langle premp_2 \rangle\!\rangle.w_i} \Big) \vee$$
$$\Big(\langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle premp_2 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i,\, w_{i2}=\langle\!\langle eqp \rangle\!\rangle.w_{i2}} \Big) \vee$$
$$\Big(\langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle eqp \rangle\!\rangle.w_{i2},\, w_{i2}=\langle\!\langle premp_2 \rangle\!\rangle.w_i} \Big) \vee$$
$$\Big(\langle\!\langle eqp_1 \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge \langle\!\langle eqp_2 \rangle\!\rangle_{w_{i1}=\langle\!\langle premp_2 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle eqp_1 \rangle\!\rangle.w_{i2},\, w_{i2}=\langle\!\langle eqp_2 \rangle\!\rangle.w_{i2}} \Big) \Big]$$

$(r_{12.3})$ for *rp* an *attack* relation.

$$\langle\!\langle \textbf{\textit{attack}} \rangle\!\rangle = \langle\!\langle rebut \rangle\!\rangle \; UNION \; \langle\!\langle undercut \rangle\!\rangle$$

$(r_{12.4})$ for *rp* an *endorse* relation.

$$\langle\!\langle \textbf{\textit{endorse}} \rangle\!\rangle = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \Big[(\langle\!\langle ap_1 \rangle\!\rangle.w_{ra} \; conclusion \; \langle\!\langle cp_2 \rangle\!\rangle.w_i) \;\vee$$
$$(\langle\!\langle ap_2 \rangle\!\rangle.w_{ra} \; conclusion \; \langle\!\langle cp_1 \rangle\!\rangle.w_i) \;\vee\; \langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i; w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \Big]$$

$(r_{12.5})$ for *rp* a *back* relation

$$\langle\!\langle \textbf{\textit{back}} \rangle\!\rangle = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \Big((\langle\!\langle ap_2 \rangle\!\rangle.w_{ra} \; premise \; \langle\!\langle cp_1 \rangle\!\rangle.w_i) \;\vee$$
$$\langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra} \; premise \; \langle\!\langle eqp \rangle\!\rangle.w_{i2}) \Big)$$

$(r_{12.6})$ for *rp* a *support* relation

$$\langle\!\langle \textbf{\textit{support}} \rangle\!\rangle = \langle\!\langle endorse \rangle\!\rangle \; UNION \; \langle\!\langle back \rangle\!\rangle$$

(r_{13}) *pathpattern pp* between two argument patterns $ap_1 =< pr_1, cp_1 >$ and $ap_2 =< pr_2, cp_2 >$

$(r_{13}.1)$ for *pp* a relation pattern r $\quad \langle\!\langle pp \rangle\!\rangle = \langle\!\langle r \rangle\!\rangle$

$(r_{13}.2)$ for $pp = pp_1/pp_2$, where pp_1, pp_2 pathpatterns, we have:

$$\langle\!\langle pp_1/pp_2 \rangle\!\rangle = \langle\!\langle pp_1 \rangle\!\rangle \wedge \langle\!\langle pp_2 \rangle\!\rangle_{\langle\!\langle ap_1 \rangle\!\rangle.w_{ra}=\langle\!\langle pp_1 \rangle\!\rangle.\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}}$$

$(r_{13}.3)$ for $pp = pp * n$, written as $pp/\ldots/pp$ (n repetitions):

if $n = 1$, then $\langle\!\langle pp{*}n \rangle\!\rangle = \langle\!\langle pp \rangle\!\rangle$

if $n > 1$, then $\langle\!\langle pp{*}n \rangle\!\rangle = \langle\!\langle pp \,/\, pp{*}(n\text{-}1) \rangle\!\rangle$

Table 4.4: ArgQL to SPARQL Translation

(r_{13}) *pathpattern pp* between two argument patterns $ap_1 =< pr_1, cp_1 >$ and $ap_2 =< pr_2, cp_2 >$

$(r_{13.4})$ for pp = pp + n, which means, "at most n repetitions of pp":

$$\langle\langle pp{+}n \rangle\rangle = \bigcup_{k=1}^{n}(pp * k) = \langle\langle pp{*}1 \rangle\rangle \textit{ UNION .. UNION } \langle\langle pp{*}n \rangle\rangle$$

(r_{14}) for dialogue pattern dp:

if $dp = ap$, where ap an argument pattern, then: $\langle\langle dp \rangle\rangle = \langle\langle ap \rangle\rangle$

if $dp = ap\ pp\ dp'$, with *head*(dp') the first argument pattern appearing in the sequence defined by dp', we have:

$$\langle\langle dp \rangle\rangle = \langle\langle ap \rangle\rangle \wedge \langle\langle pp \rangle\rangle_{\langle\langle ap_1 \rangle\rangle.w_{ra} = \langle\langle ap_0 \rangle\rangle.w_{ra}\ ;\ \langle\langle ap_2 \rangle\rangle.w_{ra} = \langle\langle head(dp') \rangle\rangle.w_{ra}} \wedge \langle\langle dp' \rangle\rangle$$

Table 4.5: ArgQL to SPARQL Translation

be exactly equal to this set.

Example 4.1.1. *Assuming that the premise pattern premp of an argument pattern is the set* $\{'p_1','p_2'\}$, *we have that:*

```
⟪premp⟫ = (?i1 rdf:type aif:I-node). (?i1 aif:claimText 'p'1) .                 (⟪'p1'⟫)
{ (?i1 aif:premise ?ra)}
UNION {(?i1 rdf:type aif:I-node). (?i1 aif:claimText 'p'1)
(?ya1 aif:illocutionaryContent ?i1).(?ya1 rdf:type YA-node).
(?ya1 aif:locution ?loc1).(?loc1 rdf:type L-node).  (?i2 rdf:type aif:I-node).
(?i2 aif:claimText i3).(?ya2 rdf:type aif:YA-node).(?ya2 aif:illocutionaryContent ?i2).
(?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 rdf:type aif:YA-node).(?ya3 aif:anchor ta1).
(?ya3 aif:illocutionaryContent ?ma1).(?ma1 rdf:type MA-node).                     (⟪ eq1⟫)
(?i2 aif:premise ?ra)}
(?i4 rdf:type aif:I-node).(?i4 aif:claimText 'p'2).                              (⟪'p2'⟫)
{ (?i4 aif:premise ?ra)}
UNION { (?i4 rdf:type aif:I-node).(?i4 aif:claimText 'p'2).
(?ya4 aif:illocutionaryContent ?i4).(?ya4 rdf:type YA-node).
(?ya4 aif:locution ?loc3).(?loc3 rdf:type L-node).  (?i5 rdf:type aif:I-node).
(?i5 aif:claimText i6).(?ya5 rdf:type aif:YA-node).  (?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).
(?ya6 rdf:type aif:YA-node).(?ya6 aif:anchor ta2).
(?ya6 aif:illocutionaryContent ?ma2).(?ma2 rdf:type MA-node).                     (⟪eq2 ⟫)
```

$(?i_5$ `aif:premise` $?ra)\}$

FITER NOT EXISTS $\{$ **FILTER**$(?x$ `!=` $?i_1)$. **FILTER**$(?x$ `!=` $?i_4)$. $(?x$ `aif:Premise` $?ra)$ $\}$ $(\langle\langle$***premp***$\rangle\rangle)$

Considering the $r_{8.2}$, in correspondence with ArgQL where the part v_p will match with the set of arguments' premises, its translation will express that w_is will bind with I-nodes which will be premises of the w_{ra}. Note that variable v_p from ArgQL is reused, so that it will get the string values of the property *claimText*. The filter part f_{pr} is translated separately, but it has to join on the variable w_{ra} such that any restrictions in the filter will pertain to the particular *ra*.

We will now discuss about r_9 which defines the translation of the different types of premise *filters*. Rule $r_{9.1}$ translates the filter that requires a particular set to be included in the premises of the required argument. That case is reduced to the rule $r_8.1$, since we also search here whether any equivalents of the propositions are included in the premises. Rule $r_{9.2}$ translates the filter that demands *all* of the premises of another argument to be included in the current one. SPARQL does not provide a way to express the *"for all"* case directly. To avoid this problem, we paraphrase it and write the query as: "there is no I-node in the premises of the matching RA-node of $\langle\langle premp_2\rangle\rangle.w_{ra}$, which is not included in the premises of the matching RA-node of the current w_{ra}". The translation must also take into account the case in which, the two sets do not include the same, but equivalent propositions and this is shown in the last two cases of the disjunction. Rule $r_{9.3}$ translates the filter in which at least one of the propositions has to be included in the premises of the required arguments. This is expressed as a disjunction between the individual proposition patterns of the given set. The rest is similar with the rule $r_{9.1}$, except from the fact that, the disjunction here means that at least one of the given propositions (or some equivalent) must be included in the premises of the required argument. Finally, rule $r_{9.4}$ requires a join between the premise patterns or their equivalents.

Rule r_{10} translates the conclusion pattern. Again here we check separately its two forms, that is, the case of a constant proposition pattern, or a variable respectively. The first case is shown in the rule $r_{10.1}$ and the translation makes a join with the equivalence pattern in order to check whether there is an equivalent I-node that is a conclusion to the particular ra. The second form of the pattern is shown in the rule $r_{10.2}$. Here a new variable is created for the I-node uri, while variable v_c which comes from the ArgQL query will get the value of claimText property of the matching I-nodes.

Rule r_{11} translates a complete argument pattern. The resulted graph pattern is simply a join of the translations of the respective premise and conclusion patterns on the variable w_{ra}. Note that when the argument pattern is a single variable it is translated as if it was the argument pattern $\langle ?pr, ?c\rangle$.

Example 4.1.2. *Let the argument pattern* $ap = \langle ?pr[/?pr_2],' cval'\rangle$. *We assume that it has been verified by the parser, that variable* $?pr_2$ *belongs to the premise pattern of a different argument pattern and that this argument pattern has been translated and has been assigned the variable* $?ra'$ *for the respective RA-node. We have that:*

$\langle\langle$**ap**$\rangle\rangle$ = $(?i_1$ `rdf:type aif:I-node`$).(?i_1$ `aif:claimText` $?pr)$.

$(?i_1$ `aif:premise` $?ra)$

$\{$**FILTER NOT EXISTS** $\{$ $(?x$ `aif:Premise` $?ra')$

```
FILTER NOT EXISTS { (?x aif:premise ?ra) } }
} UNION {
FILTER NOT EXISTS { (?x aif:Premise ?ra').
(?ya1 aif:illocutionaryContent ?x).(?ya1 rdf:type YA-node).
(?ya1 aif:locution ?loc1).(?loc1 rdf:type L-node).(?i2 rdf:type aif:I-node).
(?i2 aif:claimText i4).(?ya2 rdf:type aif:YA-node).(?ya2 aif:illocutionaryContent ?i2).
(?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 rdf:type aif:YA-node).
(?ya3 aif:anchor ta1).(?ya3 aif:illocutionaryContent ?ma1).
(?ma1 rdf:type MA-node).
FILTER NOT EXISTS { ?i2 aif:Premise ?ra) } }
}                                                                            (⟪fpr⟫)
{(?i3 rdf:type aif:I-node).(?i3 aif:claimText 'cval').                      (⟪'cval'⟫)
( ?ra aif:conclusion ?i4 ) }
UNION { (?ya4 aif:illocutionaryContent ?i3).(?ya4 rdf:type YA-node).
(?ya4 aif:locution ?loc3).(?loc3 rdf:type L-node).(?i4 rdf:type aif:I-node).
(?i4 aif:claimText i5).(?ya5 rdf:type aif:YA-node).(?ya5 aif:illocutionaryContent ?i4).
(?ya5 aif:locution ?loc4).(?loc4 rdf:type aif:L-node).(?ta2 rdf:type aif:TA-node).
(?loc4 aif:startLocution ?ta2).  (?ta2 aif:endLocution loc4).(?ya6 rdf:type aif:YA-node).
(?ya6 aif:anchor ta2).(?ya6 aif:illocutionaryContent ?ma2).
(?ma2 rdf:type MA-node).                                                     (⟪ eqp ⟫)
(?ra aif:conclusion ?i3) }                                                   (⟪ conclp ⟫)
(?ra rdf:type RA-node) .                                                     (⟪ ap⟫)
```

Rule r_{12} translates the *relation* pattern. The ArqQL semantics of this pattern type say that, for example a *rebut* relation will match any path $P_{a_1 \to a_2} = (a_1 \textit{ rebut } a_2)$ existing in the data. This feature must also be preserved in their translation and it justifies the decision why the two argument patterns on the left and right side must be arbitrary. Each of the four types generates a different graph pattern, thus they are examined separately. *Rebut* relation is translated by rule $r_{12.1}$. The translation asks for matching conflicts (CA) between the two I-nodes which are conclusions of the ra_1 and ra_2 that constitute matches of ap_1 and ap_2 respectively, or between their equivalents. As a result, we create two joins between the two conclusion patterns and the equivalence pattern to retrieve their equivalents, and then, with the four unions, we check whether there is some ca instance among any of the four combinations. *Undercut* relation pattern is translated by the rule $r_{12.2}$. Here, the generated graph pattern asks for conflicts between cp_2 and the general I-node variable that corresponds to the premise pattern v_p of ap_2 or some equivalent to it. The translation of *attack* is then defined as the union of the *rebut* and *undercut* patterns (rule $r_{12.3}$). Rule $r_{12.4}$ translates the *endorse* relation. In particular it checks whether cp_1 or its equivalent is also conclusion to ap_2 or vice versa, whether cp_2 or its equivalent is also conclusion to ap_1. About *back* relation ($r_{12.5}$), the generated graph pattern must check whether the conclusion of ap_1 or its equivalent belongs at the same time, to the premises of ap_2. The translation of *support* is then defined as the union of the *endorse* and *back* patterns (rule $r_{12.6}$).

Rule r_{13} realizes the translation of path patterns. The first sub-rule $r_{13.1}$ which refers to the simplest case that the path pattern is a single relation, the translation of the path pattern is equal to the graph pattern generated by the particular relation pattern. Rule $r_{13.2}$ captures the case in which the path pattern is a sequence of relation patterns. Thus, the generated graph pattern is the conjunction of the translation of the individual relations in the sequence, with the additional constraint, that the w_{ra} variable of the ap_1 from pp_2 must be the same with the w_{ra} variable of the ap_2 from pp_1. The reason for preferring that way over the choice of translating it directly to the respective property path of SPARQL1.1 [52], is that we are able to retrieve the values for the intermediate arguments and, based on these values, we can build and return the paths in the final query. In the alternative choice we wouldn't have this capability. Rule $r_{13.3}$ is just a different way to write a sequential path pattern, so it holds whatever we discussed in case r_{13}.2. About the rule $r_{13.4}$, the symbol $+n$ expresses all the different paths with length from 1 to n. So it is translated as the union of the translations of the individual path patterns.

Rule r_{14} translates complete dialogue patterns, as the conjunction of the sequences of alternations between the translated argument patterns and path patterns. Here we have to add the restriction that the first and last argument patterns of the sequence defined by the path pattern pp, must be the ones appearing in the query, so we impose equality in the respective w_{ra} variables.

Finally, taking into account that the general form of a complete ArgQL query is:

$$\texttt{q} \leftarrow \textbf{match}\ \texttt{dp}_1\ \texttt{, ..., dp}_\texttt{n}\ \textbf{return}\ \texttt{varlist}$$

The equivalent SPARQL of q, denoted as $q* \in Q_S$, where Q_S is the infinite set of SPARQL queries has the following general form:

$$q* \leftarrow \texttt{Select * Where} \bigwedge_{i=1}^{n} \langle\langle dp_i \rangle\rangle$$

We use the *select* * because during the translation process, new variables have been created the values of which are later necessary to transform the results back to the expected format.

In the example 4.1.3 we show the translation of a complete ArgQL query. We use that example, not so much for the understanding of the translation, but basically to emphasize the expressive usefulness of ArgQL and also to verify our initial claim that, the same query may be quite struggling to be written and understood with one of the standard languages and much easier with ArgQL.

Example 4.1.3. *Let the ArgQL query*

```
q ← match ?a1 attack/attack ?a2:  < ?pr1, 'cval' > ,
    ?a3:<?pr2[/?pr1], ?c>
    return ?a1, ?a3
```

which asks for arguments that are of distance of two attack relations from arguments with conclusion 'cval', and at the same time, those arguments the premises of which is a superset of the arguments' *premises which have the conclusion 'cval'. The query includes three argument patterns, let ap_1, ap_2 and ap_3, that correspond to the three variables $?a_1, ?a_2, ?a_3$ respectively, and the path pattern (pp) attack/attack. We also imply the argument pattern ap_x that will match to the intermediate arguments between the two attack relations in the sequence. The SPARQL equivalent $q*$ is given below:*

```
prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix aif:<http://www.arg.dundee.ac.uk/aif#>
SELECT *
FROM <http://aif.db>
WHERE {
(?i1 rdf:type aif:I-node).(?i1 aif:claimText ?txt1).(?i1 aif:Premise ?ra1).
(?i2 rdf:type aif:I-node).  (?i2 aif:claimText ?txt2).  (?ra1 aif:Conclusion ?i2).
(?ra1 rdf:type aif:RA-node).                                                  ((⟪ap1⟫))
(?i5 rdf:type aif:I-node).(?i5 aif:claimText ?pr1).(?i5 aif:Premise ?ra3).
(?i6 rdf:type aif:I-node).(?i6 aif:claimText 'cval').
{ (?ra3 aif:Conclusion ?i6).  }
UNION { (?ya1 aif:IllocutionaryContent ?i6).(?ya1 aif:Locution ?loc1 ).
(?loc1 rdf:type aif:L-node).(?i7 rdf:type aif:I-node).(?i7 aif:claimText ?txt5).
(?ya2 aif:IllocutionaryContent ?i7).(?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).
(?ra3 aif:Conclusion ?i7).}
(?ra3 rdf:type aif:RA-node).                                                  ((⟪ap2⟫))
(?i3 rdf:type aif:I-node).(?i3 aif:claimText ?txt3).(?i3 aif:Premise ?ra2).
(?i4 rdf:type aif:I-node).(?i4 aif:claimText ?txt4).(?ra2 aif:Conclusion ?i4).
(?ra2 rdf:type aif:RA-node).                                                  ((⟪apx⟫ )
{ (?ca1 rdf:type aif:CA-node).(?i2 aif:Premise ?ca1).(?ca1 aif:Conclusion ?i4).
} UNION { (?ca1 rdf:type aif:CA-node).(?i2 aif:Premise ?ca1).
(?ca1 aif:Conclusion ?i3).  }                                           ((⟪attack1⟫))
{ (?ca2 rdf:type aif:CA-node).(?i4 aif:Premise ?ca2).(?ca2 aif:Conclusion ?i6).
} UNION { (?ca2 aif:Conclusion ?i7).} } UNION { (?ca2 rdf:type aif:CA-node).
(?i4 aif:Premise ?ca2).(?ca2 aif:Conclusion ?i5).  }               ((⟪attack/attack⟫))
(?i8 rdf:type aif:I-node).(?i8 aif:claimText ?pr2).(?i8 aif:Premise ?ra4).
{ FILTER NOT EXISTS { ?x aif:Premise ?ra3) FILTER NOT EXISTS { ?x aif:Premise ?ra4) } }
} UNION {
FILTER NOT EXISTS { ?x aif:Premise ?ra3).
(?ya4 aif:illocutionaryContent ?x).(?ya4 rdf:type YA-node).(?ya4 aif:locution ?loc3).
(?loc3 rdf:type L-node).(?i9 rdf:type aif:I-node).(?i5 aif:claimText i10).
```

```
(?ya5 rdf:type aif:YA-node).(?ya5 aif:illocutionaryContent ?i9).(?ya5 aif:locution ?loc4).
(?loc4 rdf:type aif:L-node).(?ta2 rdf:type aif:TA-node).(?loc4 aif:startLocution ?ta2).
(?ta2 aif:endLocution loc4).(?ya6 rdf:type aif:YA-node).(?ya6 aif:anchor ta2).
(?ya6 aif:illocutionaryContent ?ma2).(?ma2 rdf:type MA-node).
FILTER NOT EXISTS { ?i9 aif:Premise ?ra4) } }                              ⟪pr2[/?pr1]⟫
} FILTER (?ra3! =?ra4)
(?i9 rdf:type aif:I-node).(?i9 aif:claimText ?c).(?ra4 aif:Conclusion ?i9).
(?ra4 rdf:type aif:RA-node).}                                               (⟪ap3⟫)
```

We prove that the ArgQL-to-SPARQL translation is *sound and complete* with respect to the semantics of ArgQL defined in section 3.3.3 and the semantics of SPARQL of section 2.3.2 in the following theorem:

Theorem 8. *Given A an argument base, M the set with the available patterns of ArgQL, $\|\cdot\|$ and $\langle\!\langle\cdot\rangle\!\rangle$ the mappings for the data model and ArgQL patterns respectively, $I_A(\cdot)$ the interpretation function of ArgQL patterns in the argument base A and $\mathcal{E}_G(\cdot)$ the evaluation function of SPARQL graph patterns in an RDF graph G, we have that for any ArgQL pattern $m \in M$, it holds that*
$\|I_A(m)\| = \mathcal{E}_{\|A\|}(\langle\!\langle m\rangle\!\rangle)$

Intuitively, the theory says that, if we apply the SPARQL semantics on the graph pattern $\langle\!\langle m\rangle\!\rangle$ generated by the translation of a particular ArgQL pattern m, against the translated argument base $\|A\|$, we will get the same results with those generated by translating the data that resulted from the application of ArgQL semantics on m against the original argument base ($\|I_A(m)\|$). The proof of the theorem is given in the appendix .2. We give here a sketch of the proof: for all the different types of m and for all the different forms each type may get, we must show that both $\|I_A(m)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle m\rangle\!\rangle)$ and $\mathcal{E}_{\|A\|}(\langle\!\langle m\rangle\!\rangle) \subseteq \|I_A(m)\|$ are true.

4.2 Optimization of the ArgQL-to-SPARQL translation

The proposed translation of section 4.1.2 is essentially a translation of ArgQL semantics into SPARQL combined with the definition of data mappings of section 4.1.1. Although in a theoretical level, the particular approach makes it straightforward to prove soundness and completeness, when applying the rules of tables 4.2-4.5 on the algorithms that implement this translation, we realize that it results to non optimal queries. With the term optimal we mean the minimum possible number of conjunctions and joins required to express the particular information requirement expressed by the query. Increments in the length of the query, cost in performance of the final query. Therefore, we have to find ways to optimize the generated query. In this section, we detect the points within the process that are able to accommodate improvements, and we suggest some ways of optimization that reduce the length of the generated SPARQL query, but without affecting the correctness of the results.

4.2.1 Truncating redundant joins

Our first observation is that, during the translation process, there is a point, for which, the defined translation rules result to redundant joins in the final query, which are useless for the location of the correct information. That point is the case of path patterns. According to the semantics of ArgQL, a path pattern matches all paths consisted of the particular relation sequence, regardless of which is the first or last argument. To translate this in SPARQL, we temporarily disregard the argument patterns ap_1 and ap_2 explicitly appearing in the query, and we assume that the path pattern lies between two random (general) argument patterns, let ap'_1 and ap'_2. These random argument patterns are translated separately, and the graph patterns they generate, join later with the translated ap_1 and ap_2 on the respective w_{ra} variables. ($\langle\!\langle ap_1 \rangle\!\rangle$ with $\langle\!\langle ap'_1 \rangle\!\rangle$ and $\langle\!\langle ap_2 \rangle\!\rangle$ with $\langle\!\langle ap'_2 \rangle\!\rangle$). The optimization we suggest here, is to detour the assumption of the random argument patterns and translate directly the ap_1 and ap_2, such that the translated path pattern will find only those paths for which the starting and ending arguments are matches of ap_1 and ap_2. It is easy to understand intuitively and also show formally that, withdrawing the particular parts in the query, has no effect in the matching results.

4.2.2 Dealing with the equivalence semantics issues

The second point we detect that largely afflicts the efficiency of the query, is when querying about equivalence. This happens for two main reasons.

First of all, the equivalence factor contributes largely to the length of the final SPARQL query. We already saw that each equivalence pattern is translated to 17 triple patterns (rule r_6), and since there are many points in the query in which the equivalence has to be checked, it is inevitable that the particular piece will be repeated several times in the SPARQL query. Furthermore, it also generates a wide number of unions, in order to check for the different cases of equivalence. To give an indicative example of this number, we will show the worst such case which is the *attack* relation. As shown in the rule $r_{12.3}$, an attack relation can be either a *rebut*, or an *undercut* and by the rules $r_{12.1}$ and

$r_{12.2}$, we see that each of them uses 4 unions to catch all possible pattern types taking also into account the equivalences. Now consider the path pattern *(attack)+n*, which asks for all the different paths of length *1* to *n* consisting of the attack relation. The translation of only this simple path pattern of the ArgQL query, uses *9*n* unions (e.g. the *(attack)+3* uses 27) in which the translation of the equivalence pattern appears several times. It is easy to imagine how much the length increases with any new component inserted to the ArgQL query (we show such representative numbers in the experimental evaluation of section 5.2). In all these joins and unions, the number of variables proliferates analogously, which makes the execution of this SPARQL query extremely slow, and sometimes even unable to be executed or process the results. In addition, the wide number of unions and joins, toughens the comprehensibility of the generated query, as well.

The second problem with the current approach is that, the syntax of SPARQL does not allow to express the transitivity property of the equivalence relation. For example we know from the theory of section 3.1 that if $x \equiv y$ and $y \equiv z$, then $x \equiv z$. With SPARQL, there is no mechanism to express this inference, unless we write it explicitly in the query, which was initially our suggestion as shown in the translation tables above. In this way though, we only capture limited transitivity steps and in particular only as many as written in the query. This creates the risk of losing information by ignoring existing equivalences in the translated argument base.

In order to resolve the aforementioned problems, we were inspired by the approach proposed in [123]. The general idea is that, the values of the propositions shift to a different range and are assigned values from a set C. According to this new assignment, which is called *canonical assignment*, it holds that all *equivalent* propositions in $\mathcal{L}$ share a common value, called *canonical identifier*. In the RDF representation, these identifiers replace the values of the respective I-nodes.

Canonical assignment is formally defined as follows:

Definition 24 (Canonical assignment)**.** *Given C a set of canonical identifiers, the canonical assignment is defined as the function $can : \mathcal{P} \mapsto C$ such that $can(p_1) = can(p_2)$ if and only if $p_1 \equiv p_2$.*

With this shift to the range C, any time we want to query about equivalent propositions, it suffices to search by their canonical identifiers. The problems stated above, basically regard the effectiveness of the proposed methodology for execution and in particular, are caused by the defined translation and from the expressiveness limitations of SPARQL. Therefore, in the remainder of this chapter, we are going to show how this idea is incorporated into this process and in particular, which parts of the data mappings and query translation are affected, and how.

This approach constitutes a solution to the problems stated above. Regarding the first problem, as it will become apparent later in section 5.2.2, the size of the generated query is significantly diminished after this optimization. This happens because all those pieces of the query that correspond to the equivalence pattern, as well as all of the relevant unions, will be truncated from the SPARQL query and will be replaced by simple equality conditions of the canonical elements. In addition, the transitivity issue described in the second problem is also resolved, since all of the equivalent propositions share a common *canonical identifier*. Thus, searching by this for the satisfaction of some condition, it will succeed if it becomes true for any of the equivalents.

Data and query mappings under canonical assignment

We now show how the canonical assignment idea is incorporated in the process of data mapping and query translation and that this incorporation preserves the properties for soundness and completeness.

Regarding the data mapping, the canonical assignment replaces the value of a proposition p with $can(p)$, therefore it only affects the mapping rule r_1 of the table 4.1 that concerns the case of proposition. Generally, we use the notation $\|x\|^c$ to denote the mapping that takes into account the canonical assignment. In the case of proposition the data mapping becomes:

$$\|p\|^c = (u(\hat{p})\ \texttt{rdf:type aif:I-node}),\ (u(\hat{p})\ \texttt{aif:claimText}\ can(p))$$

The mapping for the rest of the cases remains the same, except the last one, which is totally omitted, since the information about equivalences is now kept in the canonical identifiers. For these cases, we have that $\|x\|^c = \|x\|$.

Table 4.6 shows how the ArgQL-SPARQL translation is modified. We rewrite the rules of the tables 4.2, 4.3, 4.4 and 4.5 that are affected, using intonation for their new version. Notation $\langle\!\langle x\rangle\!\rangle^c$ is used to indicate the translation that takes into account the canonical assignment. For the rest of the rules that do not appear in the table, we assume that $\langle\!\langle x\rangle\!\rangle^c = \langle\!\langle x\rangle\!\rangle$. The idea is that we eliminate the equivalence pattern from everywhere it appears in the process. The search is only performed by the canonical identifier and, after the data have been matched, the content of the propositions that satisfy the various patterns can be found in the database table in which they are saved, using their retrieved URI.

In the first rule r'_7, instead of using the initial value of the proposition pattern, we use the *canonical identifier*. In rule $r'_{8.1}$ the parts that search for equivalence propositions with the ones defined by the proposition patterns prp_j, being premises of the matching w_{ra} are totally eliminated with respect to $r_{8.1}$. The same holds for $r'_{9.1}$, $r'_{9.2}$,$r'_{9.3}$ and $r'_{9.4}$. In rule $r'_{10.1}$ the part that asks for equivalent propositions being conclusion of the matching w_{ra} is omitted. Rules $r'_{12.1}$ and $r_{12.2}$ are based on the idea that if there is a conflict between two particular canonical identifiers, it is inferred that the conflict holds for all of the equivalents, as well. Thus, for the case of rebut, the conflict regard the canonical identifiers of the two conclusions, while in the case of undercut, the conflict concerns the conclusion of the first and some premise of the second argument patterns. Finally, the rules $r_{12.4}$ and $r_{12.5}$ expresses that it suffices the two conclusions or the conclusion of the first with the premise of the second argument pattern to have the same canonical identifier, in order to have an endorse, or back relation, respectively.

The following example shows the translation with the canonical assignment of the query in the example 4.1.3. With this example we want to emphasize the degree of improvement that the underlying approach causes to the length of the generated query.

(r'_7) $\langle\langle \textbf{prp} \rangle\rangle^c$ = $(w_i$ *type I-node*$) \wedge (w_i$ *claimText can(p)*$)$

$$(r'_{8.1})\ \langle\langle \textbf{premp} \rangle\rangle^c = \bigwedge_{j=1}^{n} \Big(\langle\langle prp_j \rangle\rangle^c \wedge (\langle\langle prp_j \rangle\rangle^c.w_i \textit{ premise } w_{ra}) \Big) \wedge$$

$$\textit{NOT EXISTS } \{ (\bigwedge_{j=1}^{n} w_x \neq \langle\langle prp_j \rangle\rangle^c.w_i) \wedge (w_x \textit{ premise } w_{ra}) \}$$

$$(r'_{9.1})\ \langle\langle \textbf{fpr} \rangle\rangle^c = \bigwedge_{j=1}^{m} \Big(\langle\langle sp_j \rangle\rangle^c \wedge (\langle\langle sp_j \rangle\rangle^c.w_i \textit{ premise } w_{ra}) \Big)$$

$$(r'_{9.2})\ \langle\langle \textbf{fpr} \rangle\rangle^c = \textit{NOT EXISTS } \{ (w_i \textit{ premise } \langle\langle premp_2 \rangle\rangle^c.w_{ra}) \wedge$$
$$\textit{NOT EXISTS } \{ (w_i \textit{ premise } w_{ra}) \} \}$$

$$(r'_{9.3})\ \langle\langle \textbf{fpr} \rangle\rangle^c = \bigvee_{j=1}^{m} \Big(\langle\langle sp_j \rangle\rangle^c \wedge (\langle\langle sp_j \rangle\rangle^c.w_i \textit{ premise } w_{ra}) \Big)$$

$(r'_{9.4})$ $\langle\langle \textbf{fpr} \rangle\rangle^c = (\langle\langle premp_2 \rangle\rangle^c.w_i \textit{ premise } w_{ra})$

$(r'_{10.1})$ $\langle\langle \textbf{conclp} \rangle\rangle^c = \langle\langle prp \rangle\rangle^c \wedge (w_{ra} \textit{ conclusion } \langle\langle prp \rangle\rangle^c.w_i)$

$(r'_{12.1})$ $\langle\langle \textbf{rebut} \rangle\rangle^c = \langle\langle ap_1 \rangle\rangle^c \wedge \langle\langle ap_2 \rangle\rangle^c \wedge (w_x \textit{ claimText } \langle\langle cp_1 \rangle\rangle^c.val)) \wedge$
$(w_y \textit{ claimText } \langle\langle cp_2 \rangle\rangle^c.val) \wedge \langle\langle conflp \rangle\rangle^c{}_{w_{i1}=w_x,\, w_{i2}=w_y}$

(Here the $\langle\langle cp \rangle\rangle^c$.val refers to the object of *claimText* property of the main I-node of cp, and can be either a variable, or the constant value of the canonical identifier, depending on the particular form of cp.

$(r'_{12.2})$ $\langle\langle \textbf{undercut} \rangle\rangle^c = \langle\langle ap_1 \rangle\rangle^c \wedge \langle\langle ap_2 \rangle\rangle^c \wedge (w_x \textit{ claimText } \langle\langle cp_1 \rangle\rangle^c.val)) \wedge$
$(w_y \textit{ claimText } \langle\langle premp_2 \rangle\rangle^c.val)\ \langle\langle conflp \rangle\rangle^c{}_{w_{i1}=w_x,\, w_{i2}=w_y}$

$(r'_{12.4})$ $\langle\langle \textbf{endorse} \rangle\rangle^c = \langle\langle ap_1 \rangle\rangle^c \wedge \langle\langle ap_2 \rangle\rangle^c \wedge (\langle\langle cp_1 \rangle\rangle^c.val = \langle\langle cp_2 \rangle\rangle^c.val)$

$(r'_{12.5})$ $\langle\langle \textbf{back} \rangle\rangle^c = \langle\langle ap_1 \rangle\rangle^c \wedge \langle\langle ap_2 \rangle\rangle^c \wedge (\langle\langle cp_1 \rangle\rangle^c.val = \langle\langle premp_2 \rangle\rangle^c.val)$

Table 4.6: Query Translation with canonical assignment

Example 4.2.1. *Let the ArgQL query of the example 4.1.3. The SPARQL equivalent q⋆ is the following:*

```
prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix aif:<http://www.arg.dundee.ac.uk/aif#>
SELECT *
```

```
FROM <http://aif.db>
WHERE {
(?i1 rdf:type aif:I-node).(?i1 aif:claimText ?txt1).(?i1 aif:Premise ?ra1).
(?i2 rdf:type aif:I-node).  (?i2 aif:claimText ?txt2).  (?ra1 aif:Conclusion ?i2).
(?ra1 rdf:type aif:RA-node).                                          (⟪ap1⟫^c)
(?i5 rdf:type aif:I-node).(?i5 aif:claimText ?pr1).(?i5 aif:Premise ?ra3).
(?i6 rdf:type aif:I-node).(?i6 aif:claimText can('cval')).(?ra3 aif:Conclusion ?i6).
(?ra3 rdf:type aif:RA-node).                                          (⟪ap2⟫^c)
(?i3 rdf:type aif:I-node).(?i3 aif:claimText ?txt3).(?i3 aif:Premise ?ra2).
(?i4 rdf:type aif:I-node).(?i4 aif:claimText ?txt4).(?ra2 aif:Conclusion ?i4).
(?ra2 rdf:type aif:RA-node).                                          (⟪ apx ⟫)
{ (?ca1 rdf:type aif:CA-node).(?i7 aif:claimText ?txt2).(?i8 aif:claimText ?txt4).
(?i7 aif:Premise ?ca1).(?ca1 aif:Conclusion ?i8).
} UNION { (?ca1 rdf:type aif:CA-node).(?i7 aif:claimText ?txt2).(?i8 aif:claimText ?txt3).
(?i7 aif:Premise ?ca1).(?ca1 aif:Conclusion ?i8).  }                  (⟪ attack⟫^c)
{ (?ca2 rdf:type aif:CA-node).(?i9 aif:claimText ?txt4).(?i10 aif:claimText can('cval')).
(?i9 aif:Premise ?ca2).(?ca2 aif:Conclusion ?i10).
} UNION { (?ca2 rdf:type aif:CA-node).(?i9 aif:claimText ?txt4).(?i10 aif:claimText ?pr1).
(?i9 aif:Premise ?ca1).(?ca1 aif:Conclusion ?i10).  } }               (⟪ attack/attack ⟫)
(?i11 rdf:type aif:I-node).(?i11 aif:claimText ?pr2).(?i11 aif:Premise ?ra4).
FILTER NOT EXISTS { ?x aif:Premise ?ra3)
FILTER NOT EXISTS { ?x aif:Premise ?ra4) } }                           ⟪ pr2[/?pr1]⟫
FILTER (?ra3!=?ra4)
(?i12 rdf:type aif:I-node).(?i12 aif:claimText ?c).(?ra4 aif:Conclusion ?i12).
(?ra4 rdf:type aif:RA-node).}                                         (⟪ap3⟫^c)
```

The following theorem expresses that the approach of canonical assignment does not violate the soundness and completeness of the translation process.

Theorem 9. *For any ArgQL pattern $m \in \mathcal{M}$, its translation to SPARQL under the canonical assignment method, is sound and complete, which means that:* $\|I_A(m)\|^c = \mathcal{E}_{\|A\|^c}(\langle\!\langle m \rangle\!\rangle^c)$

The proof of theorem 9 is trivial and it follows similar methodology with the proof of theorem 10. For convenience we are going to omit it in this document.

Chapter 5
Implementation and Evaluation

Contents

5.1 Implementation

In this section, we show the feasibility of ArgQL in real application domains by describing the process of its execution and the steps from the point of query composition, towards the retrieval of the respected results. In particular, after we give a general overview of the different components between which the data flow, as well as the different forms they may get (section 5.1.1), we will move to the description of the most important algorithms within the process (sections 5.1.2, 5.1.3) and finally we will present the ArgQL endpoint (section 5.1.4) which is a web-based application, which allows the testing and execution of queries on real data.

5.1.1 Overview

Figure 5.1 gives in an illustrative way the query execution cycle. Initially, the ArgQL query incurs lexicographic and syntactical recognition by the parsing component. The parser takes as input a query and transforms it in an intermediate structure, *Query Intermediate Data (QID)*, that constitutes a conceptual representation of the query and it is described later in that section. The RDF management component, takes as input that structure and also the translation rules defined in section 4.1.2 and generates the final SPARQL query q*. Then, the SPARQL query is executed, and finally, the retrieved RDF results are given to a set of algorithms to transform them back into the format expected by the initial ArgQL query, taking also into account the intermediate structure of the previous stage.

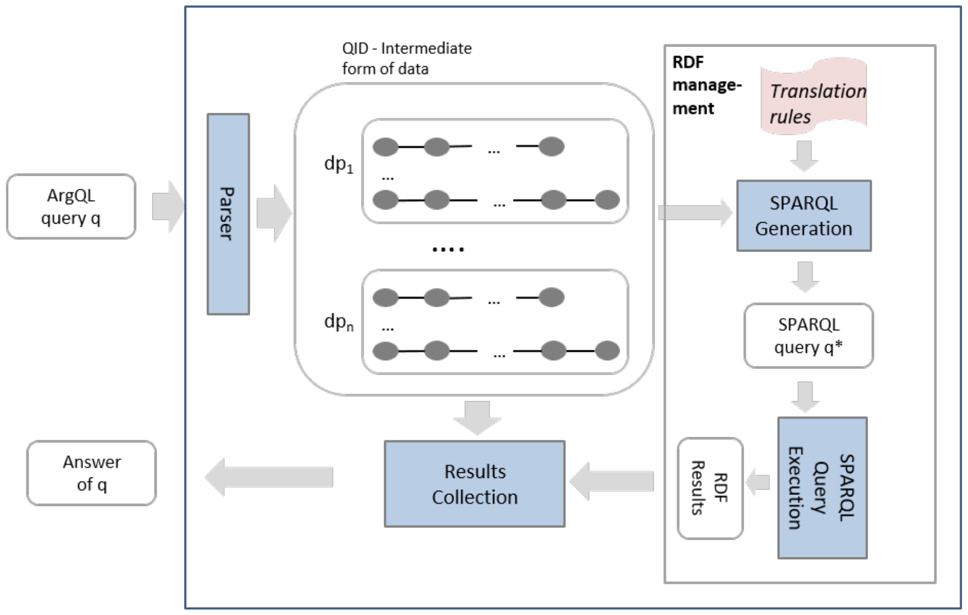
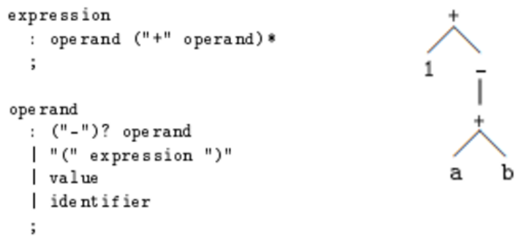


Figure 5.1: Data flow of the query execution process

5.1.2 Parsing

The ANTLR parser generator

ANother Tool for Language Recognition (ANTLR) [75] is a tool that allows for generating the parser for a specified language, from specifications in an EBNF-like syntax. The source code of the parser can be in one of the C, C++, Java, Python, C# or Objective-C programming languages. *Extended Backus-Naur Form (EBNF)* is a commonly used notation for context-free grammars. Roughly, ANTLR distinguishes three compilation phases; the lexical analysis phase, the parsing phase and tree walking phases. During the lexical analysis phase, the characters from the input file or stream are grouped into a stream of words or tokens. The token stream is used as input to the parsing phase where a tree of tokens is created, called *Abstract Syntax Trees (AST)*. An AST is a finite, labelled tree, where each node represents a programming language construct and the children of each node represent the components of that construct. Finally, a tree walker is used to process the AST that has been produced by the parser. Figure 5.2 gives an example of a EBNF grammar supported by ANTLR, as well as the AST created for a statement recognized by this grammar.



```
expression
  : operand ("+" operand)*
  ;

operand
  : ("-")? operand
  | "(" expression ")"
  | value
  | identifier
  ;
```

Figure 5.2: Example of an EBNF specification and an AST for the statement ”1 + -(a + b)”

SPARQL query generation

Before moving to the description of the parsing process, we first need to describe *QID*, the intermediate structure we referred in the previous paragraph, and then we will show how it is created by the parsing code. *QID* consists of a list of elements - *dpList* - that represent the dialogue patterns appearing in the query. Each such element dp_i in *dpList* can have two different forms: it may either be an argument pattern object, or a *ppList*, that is, a list of path patterns. In the latter case, a *ppList* represents a specific path pattern expression in the query and includes all the different alternative path patterns which are created by the combination of the * and + numeric indicators. Each path pattern pp_j in *ppList*, consists of a sequence of objects that correspond to relation patterns rp_k. Each relation pattern includes two argument patterns for the source and the destination arguments of the matching relation. In the following, we give an example of the *QID* structure, a particular ArgQL query generates.

Example 5.1.1. *Let the match clause of a query q:*
match $?a_1$ *(attack)+3* $?a_2$*:⟨ ?pr₁, ”c” ⟩,* $?a_3$*:⟨ ?pr₂[/?pr₁], ?c ⟩*

In this query we have three argument patterns $ap_1 = ?a_1$, $ap_2 = ?a_2$*:⟨ ?pr₁, ”c” ⟩,* $ap_3 = ?a_3$*:⟨ ?pr₂[/?pr₁], ?c ⟩ and a path pattern* $pp = (attack) + 3$ *between* ap_1 *and* ap_2*.*

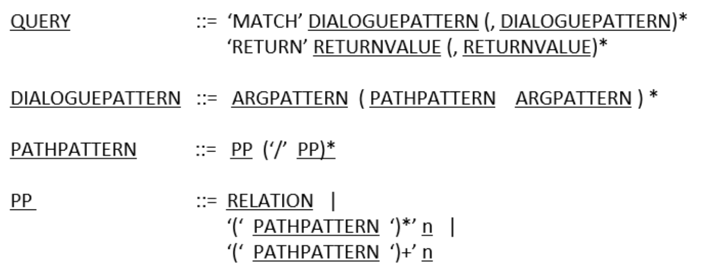
The QID structure for q consists of a dpList with two elements dp_1 *and* dp_2*. The first one,* dp_1 *will contain a ppList which will include three elements* pp_i*, that correspond to the three alternative path patterns, that pp generates. In particular:*

- $pp_1 : [ap_1\ attack\ ap_2]$
- $pp_2 : [ap_1\ attack\ \_ap_x], [\_ap_x\ attack\ ap_2]$
- $pp_3 : [ap_1\ attack\ \_ap_x], [\_ap_x\ attack\ \_ap_y], [\_ap_y\ attack\ ap_2]$

Each element in a pp_i *corresponds to a relation pattern object, which contains two argument patterns and its type (attack) and the* $\_ap_x$ *and* $\_ap_y$ *are random arguments, generated during the process (not existed in q) and will bind to the intermediate arguments in the path.*

The second one dp_2 *corresponds to the argument pattern* $?ap_3$*.*

Recall the grammatical rules shown in figure 5.3 that recognize the general form of an ArgQL query and the path pattern expressions. We restrict ourselves only to the particular set of rules, because we are not interested to overload the reader with too many technical details of the code, but to give a general idea of the generation of QID, and how it is used to create the final SPARQL query. Thus, the rules for *argpattern*, *returnvalue* and *relation* are omitted. We use capitalized and underlined text font to denote the names of the rules. Observe that the rule *pathpattern* is recursive, such that it will be able to recognize any combinations of pathpatterns, including the sequence character '/', or the numerical indicators * and +.

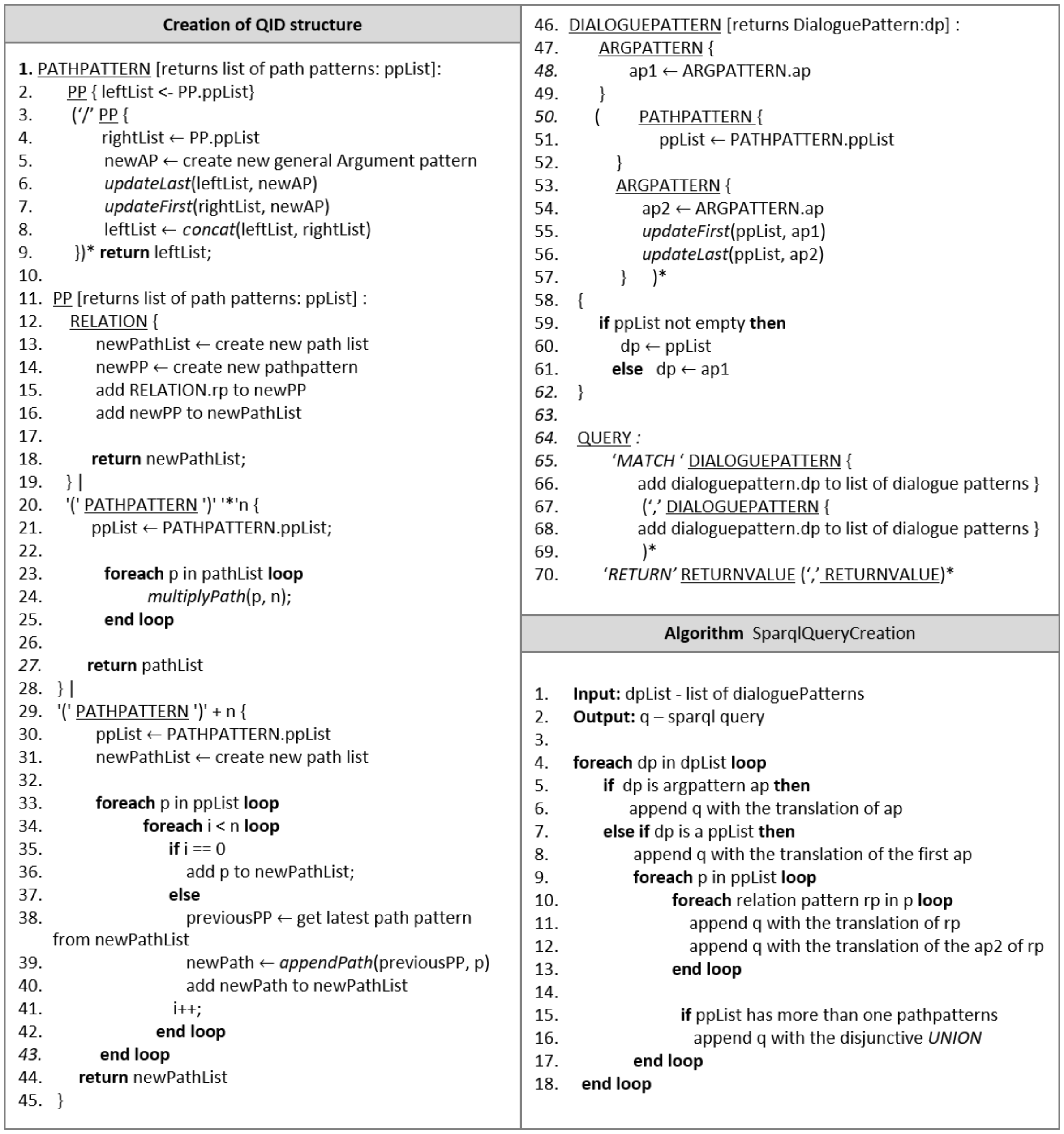


```
QUERY             ::= 'MATCH' DIALOGUEPATTERN (, DIALOGUEPATTERN)*
                      'RETURN' RETURNVALUE (, RETURNVALUE)*

DIALOGUEPATTERN   ::= ARGPATTERN ( PATHPATTERN  ARGPATTERN ) *

PATHPATTERN       ::= PP ('/' PP)*

PP                ::= RELATION  |
                      '(' PATHPATTERN ')*' n  |
                      '(' PATHPATTERN ')+' n
```

Figure 5.3: Part of the EBNF for ArgQL

To describe the parsing algorithm, we adopt the approach of ANTLR. In particular, ANTLR allows to embed code inside the EBNF language specification, in order to define the functionality triggered at the recognition of a particular construct in the specified language. Thus, in figure 5.4 we show in pseudo-code, the algorithms that are embedded in the rules of fig 5.3 and are responsible to generate QID, as well as the algorithm for the composition of the final SPARQL query from QID. Each rule returns a particular value to the parent node in the parse tree and this value is accessed with the dot(.) character by the caller. For example if a rule r_1 returns a value a and the rule r_2 uses r_1, then r_2 obtains access to the return value of r_1, with the notation $r_1.a$.

We start describing the process with the rule *pp*. Note that both *pp* and *pathpattern* rules return a ppList object to the higher levels of the parsing tree. In the simplest case in which *pp* is a simple relation pattern (line 12) a new *ppList* is defined (line 13) and then (lines 14-16) a new path pattern with a single relation pattern of the specific type is also created and added to the new ppList which is returned by the rule. If pp is of the form *(pathpattern)*n* (line 20), then all the path patterns in the ppList returned by the rule are multiplied n times. For example, in the *multiplyPath* method, if *p=attack/support* and n=3, then p will become *attack/support/attack/support/attack/support*. If pp is of the form *(pathpattern)+n* (line 29), then what must happen, is that each path pattern p in ppList returned by the *pathpattern* rule, has to generate *n* new path patterns, such that each new path pattern i will be the same as i-1 appended by p. For example, let ppList containing two path patterns *attack* and *attack/attack* and let *n=3*. The new ppList will contain 6 elements which will have arisen by the creation of the 3 new path patterns for each one existed initially in ppList.



Creation of QID structure

```
1. PATHPATTERN [returns list of path patterns: ppList]:
2.     PP { leftList <- PP.ppList}
3.      ('/' PP {
4.           rightList ← PP.ppList
5.           newAP ← create new general Argument pattern
6.           updateLast(leftList, newAP)
7.           updateFirst(rightList, newAP)
8.           leftList ← concat(leftList, rightList)
9.       })* return leftList;
10.
11.  PP [returns list of path patterns: ppList] :
12.     RELATION {
13.          newPathList ← create new path list
14.          newPP ← create new pathpattern
15.          add RELATION.rp to newPP
16.          add newPP to newPathList
17.
18.         return newPathList;
19.     } |
20.     '(' PATHPATTERN ')' '*'n {
21.         ppList ← PATHPATTERN.ppList;
22.
23.           foreach p in pathList loop
24.                  multiplyPath(p, n);
25.           end loop
26.
27.         return pathList
28.   } |
29.   '(' PATHPATTERN ')' + n {
30.          ppList ← PATHPATTERN.ppList
31.          newPathList ← create new path list
32.
33.          foreach p in ppList loop
34.                 foreach i < n loop
35.                     if i == 0
36.                        add p to newPathList;
37.                     else
38.                        previousPP ← get latest path pattern
     from newPathList
39.                        newPath ← appendPath(previousPP, p)
40.                        add newPath to newPathList
41.                      i++;
42.                   end loop
43.           end loop
44.       return newPathList
45.   }
46.  DIALOGUEPATTERN [returns DialoguePattern:dp] :
47.       ARGPATTERN {
48.            ap1 ← ARGPATTERN.ap
49.       }
50.     (       PATHPATTERN {
51.                ppList ← PATHPATTERN.ppList
52.          }
53.         ARGPATTERN {
54.              ap2 ← ARGPATTERN.ap
55.              updateFirst(ppList, ap1)
56.              updateLast(ppList, ap2)
57.          }     )*
58.   {
59.       if ppList not empty then
60.          dp ← ppList
61.        else  dp ← ap1
62.   }
63.
64.   QUERY :
65.        'MATCH ' DIALOGUEPATTERN {
66.              add dialoguepattern.dp to list of dialogue patterns }
67.              (',' DIALOGUEPATTERN {
68.              add dialoguepattern.dp to list of dialogue patterns }
69.              )*
70.       'RETURN' RETURNVALUE (',' RETURNVALUE)*
```

Algorithm SparqlQueryCreation

```
1.   Input: dpList - list of dialoguePatterns
2.   Output: q – sparql query
3.
4.   foreach dp in dpList loop
5.        if  dp is argpattern ap then
6.            append q with the translation of ap
7.        else if dp is a ppList then
8.             append q with the translation of the first ap
9.             foreach p in ppList loop
10.                 foreach relation pattern rp in p loop
11.                    append q with the translation of rp
12.                    append q with the translation of the ap2 of rp
13.                 end loop
14.
15.                  if ppList has more than one pathpatterns
16.                    append q with the disjunctive UNION
17.            end loop
18.   end loop
```

Figure 5.4: Pseudo-code for the most general parsing points and the generation of the SPARQL query

Now, we move on the description of the *pathpattern* rule, which also returns a ppList object. The rule recognizes sequences of expressions of the rule *pp*. The idea here is that, anytime that the parser recognizes the character '/', the ppList objects returned by the individual *pp* rules, must be concatenated. To do this, a new random argpattern *newAP* is created in line 5. The function *updateLast* in line 6 sets *newAP* as a terminating argument pattern in all path patterns included in the leftList (in the last relation pattern object of every path pattern in the list, *newAP* is set as a destination argpattern). Similarly, the function *updateFirst* in line 7 sets *newAP* as a starting argument pattern in all path patterns included in the rightList (in the first relation pattern object of every path pattern in the list, *newAP* is set as a source argpattern). The function concat in line 8, concatenates every element in the leftList with all elements in the rightList, thus the number of elements in the resulting ppList will be equal to the proliferation of the number of elements in the two individual lists.

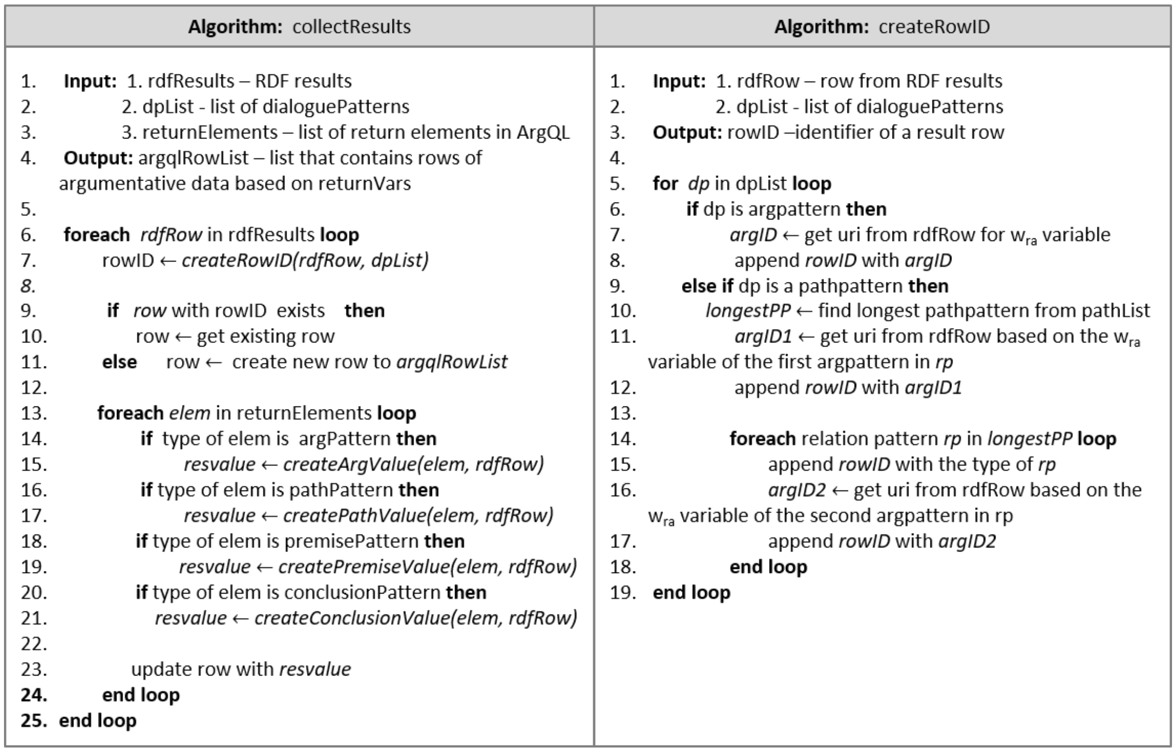
Finally, in the rules *dialoguepattern* and *query* (lines 46, 61), the final QID structure is created and in particular, ap1 is set as a starting argument pattern(line 56) in all path patterns included in ppList, returned by the rule *pathpattern*, while ap2 is set as their terminating argument pattern (line 56). Then, the dp, which is created in lines 59-61, is returned to the rule *query*, so that it will be added to the final dpList.

After having created the QID structure, it is given as input to the SparqlQueryCreation algorithm to generate the graph pattern in the *where* clause of the SPARQL query. In particular, the algorithm iterates the dpList structure and, in case that the current dp is a single argument pattern, the query is appended with its translation while if it is a ppList, it iterates the list of path patterns, and for each path pattern, it traverses the sequence of relation patterns and appends the query with the translations of the relation patterns and the intermediate argument patterns (lines 11-12), in an alternating way. If ppList has more than one path patterns, it means that they must be added as alternatives in the query, thus their translation is included in a UNION statement.

5.1.3 Results collection

In this section we describe the algorithms for the result collection and in particular, the transformation from the RDF results of the generated SPARQL query, to the form expected by the ArgQL query. The algorithms are presented in figures 5.5 and 5.6. The main algorithm is the *collectResults* in figure 5.5. The algorithm has three inputs: 1) the RDF results, which are given as rows of data that constitute the binding values of the involved RDF variables. 2) the QID structure generated during the parsing, and 3) the list with the elements in the *return* clause of the ArgQL query. An element in the return clause may be either a variable (indicating data of one of the types argument, premise, conclusion), or have the form *path*(v_1, v_2) indicating a path value. The output of the algorithm is a list of rows that will include the values of the return elements (and more precisely, all the possible combinations between the matching data of the return elements).

The general idea of the algorithm is that, the RDF data is parsed row by row (line 6) and, taking into account the QID structure, creates progressively the answer to the ArgQL query, as it is expressed in the list of return elements in the query. An important thing to mention is that, there is no one-to-one correspondence between the RDF rows and the rows for the answer of ArgQL. Instead, the data required to create an ArgQL row answer, is more likely to exist in more than one RDF rows. That possibility basically regards the case of arguments and in particular the premises of an argument. If the argument pattern contains a premise variable that is going to get the entire set of premises as its value, the number of RDF rows that will include the data for an argument value, will be equal to the number of the premises for the matching argument. As a result, for any new RDF row that is read, the algorithm either creates a new ArgQL row, or updates the already existing one with the new data.



Algorithm: collectResults

```
1.  Input: 1. rdfResults – RDF results
2.         2. dpList - list of dialoguePatterns
3.         3. returnElements – list of return elements in ArgQL
4.  Output: argqlRowList – list that contains rows of
    argumentative data based on returnVars
5.
6.  foreach rdfRow in rdfResults loop
7.      rowID ← createRowID(rdfRow, dpList)
8.
9.      if row with rowID exists then
10.         row ← get existing row
11.     else row ← create new row to argqlRowList
12.
13.     foreach elem in returnElements loop
14.         if type of elem is argPattern then
15.             resvalue ← createArgValue(elem, rdfRow)
16.         if type of elem is pathPattern then
17.             resvalue ← createPathValue(elem, rdfRow)
18.         if type of elem is premisePattern then
19.             resvalue ← createPremiseValue(elem, rdfRow)
20.         if type of elem is conclusionPattern then
21.             resvalue ← createConclusionValue(elem, rdfRow)
22.
23.         update row with resvalue
24.     end loop
25. end loop
```

Algorithm: createRowID

```
1.  Input: 1. rdfRow – row from RDF results
2.         2. dpList - list of dialoguePatterns
3.  Output: rowID –identifier of a result row
4.
5.  for dp in dpList loop
6.      if dp is argpattern then
7.          argID ← get uri from rdfRow for w_ra variable
8.          append rowID with argID
9.      else if dp is a pathpattern then
10.         longestPP ← find longest pathpattern from pathList
11.         argID1 ← get uri from rdfRow based on the w_ra
            variable of the first argpattern in rp
12.         append rowID with argID1
13.
14.         foreach relation pattern rp in longestPP loop
15.             append rowID with the type of rp
16.             argID2 ← get uri from rdfRow based on the
                w_ra variable of the second argpattern in rp
17.             append rowID with argID2
18.         end loop
19. end loop
```

Figure 5.5: Pseudo-code for the transformation between RDF results into argumentative results

In order to decide between the two choices, we maintain the idea of row ids. Each rdfRow from the RDF results is marked with an identifier, which essentially constitutes an instantiation of the dpList by the data of the rdfRow. That identifier, indicates also a particular row in the results of the ArgQL query. For each new rdfRow, we create the identifier (line 7) and if it has not been met before, a new row is created in the ArgQL results, otherwise, the existing one is retrieved in order to be updated with the data from rdfRow (lines 9-11). The returned data of the query are created in the loop between the lines 13-24. In particular, the list of the return elements is iterated, and according to whether an element is a variable of type argument, premise, conclusion, or it is the construct *path*(v_1, v_2), one of the methods *createArgValue, createPremiseValue, createConclusionValue or createPath-Value* is called to generate the respective value from data of the rdfRow and then, the result row is updated with that value.

The algorithm that creates the row identifier is also given in figure 5.5 (*createRowID*). It has two input values: a) the current rdfRow from the result set and b) the QID structure while the returned output is the row identifier. Roughly, the idea is that for each of the elements in *dpList*, if they correspond to an argument pattern ap, then we get the uri value bound to the related variable w_{ra} from the rdfRow and append the *rowID* with that value (lines 7-8), otherwise, if it includes a *ppList*, we get the last pathpattern (largest one - line 10) and we append the *rowID* alternatively, with the uri values of the w_{ra} of the intermediate argument patterns and with the character identifying the relation pattern type. The algorithm is better explained in the following example:

Example 5.1.2. *Let a dpList with two dialogue pattern objects: dp_1 defines an argument pattern ap_1 and dp_2 is created by the expression ap_2 (attack)+2 ap_3, thus it contains a ppList with two path patterns: pp_1 : (ap_2 attack ap_3) and pp_2 : (ap_2 attack $\_ap_x$ attack ap_3). Let also an rdfRow with matching RDF data of the translated dp_1 and dp_2. Assuming u_1 being the matching uri of w_{ra} of ap_1, u_2 the matching uri of w_{ra} of ap_2, u_3 the matching uri of w_{ra} of ap_3 and u_x the matching uri of w_{ra} of $\_ap_x$, the row identifier generated by the algorithm will be the "u_1, u_2Au_3" if a solution for pp_1 is found, or "$u_1, u_2Au_xAu_3$" if a solution of pp_2 is found. The letter A indicates the type of relation (attack). For example if the relation was undercut, we would use the letter U.*

Figure 5.6 shows the algorithms for the creation of an argument value *(createArgValue)* and a path value *(createPathValue)*, from an rdfRow. The two other algorithms for the premise and conclusion are trivial and we omit them here.

The algorithm *createArgValue* receives an argument pattern and an rdfRow as input and returns an argument value (more precisely, part of the argument value) created by the data in rdfRow. We assume that each value is assigned with an identifier *argID* which is created by the uri of the referred w_{ra} (line 6). Each argument pattern includes a list in which the created argument values are kept. In lines 7-9, we check whether there is an argument value with argID and if it doesn't, a new one is created, otherwise, we retrieve the existing. In the code between the lines 11-27, the premise part of the argument is created. In particular, in case that the premise pattern is a variable v, we get the respective uri and the *claimtText* value for the I-node based on the variables w_i and v (lines 12-13), respectively, a new proposition is created with those values (line 14) which is added to the set of premises of the argument value. In the other case that the premise pattern is a set of proposition patterns (line 16), that set is iterated and for each of the proposition pattern, we check whether the w_{i1} variable has value (line 18), or the w_{i2} of the equivalence pattern (line20), which means that an equivalent I-node to the one expressed in the query has been found in the premise set and the new proposition is added to the premises (line 25). Note that in this case, one single rdfRow, contains the complete value of an argument. The conclusion part of the argument is created in the code between 29-41. Again here, if the conclusion pattern is a variable v (line 29), the id and the value of the proposition are assigned to the variables w_i and v while in the case that it is a proposition pattern, it has to be checked whether there is a value for variable w_{i1} (line 34) of the w_{i2} of the equivalent pattern (line 36). Finally the *values* list is updated with the new argument value.

The algorithm *createPathValue* receives a dialogue pattern and an rdfRow as input, and returns a *path* value as output. In order to know which of the alternative path patterns in ppList of the dp is satisfied by the rdfRow, it suffices to check only the longest path pattern(line 6). In the loop between

Algorithm: createArgValue

```
1.  Input: 1. ap – Argument pattern
2.         2. rdfRow - row from RDF results
3.  Output: argValue – generated argument value
4.
5.  // values is a list of arguments which are answers to ap
6.  argID ← get uri from rdfRow based on the $w_{ra}$ variable
7.  if argID exists in values then
8.      argValue ← retrieve argument with argID
9.  else argValue ← create new argument with argID
10.
11. if premisepattern of ap is variable v then
12.     propID ← get uri from rdfRow based on variable $w_i$
13.     propValue ← get value from rdfRow based on
    variable v
14.     prop ← new proposition with propID and propValue
15.     add prop to premises of argValue.
16. else if premise pattern is prpset (set of proposition
    patterns) then
17.     for prp in prpset loop
18.         if $w_{i1}$ has value in rdfRow then
19.             prop ← new proposition with propID and
    prp value
20.         else if $w_{i2}$ has value in rdfRow then // an
    equivalent proposition found
21.             propID ← get uri from rdfRow based on $w_{i2}$
    (variable for equivalent I-node)
22.             propValue ← get text value from rdfRow
    based on text variable for the equivalent I-node
23.             prop ← new proposition with propID and
    propValue
24.         end if
25.     add prop to premises of argValue.
26. end if
27. set premises to argValue
28.
29. if conclusionpattern of ap is variable v then
30.     propID ← get uri from rdfRow based on the $w_i$
    variable
31.     propValue ← get value from rdfRow based on
    variable v
32.     prop ← new proposition with propID and propValue
33. else if conclusionpattern of ap is a proposition pattern prp
    then
34.     if $w_{i1}$ variable of prp has value in rdfRow then
35.         prop ← new proposition with propID and prp value
36.     else if $w_{i2}$ has value in rdfRow then
37.         propID ← get uri from rdfRow based on the $w_{i2}$
38.         propValue ← get text value from rdfRow based on
    text variable for the equivalent I-node
39.         prop ← new proposition with propID and propValue
40.     end if
41. set prop as a conclusion of argValue
42. end if
43. update values with argValue.
```

Algorithm: createPathValue

```
1.  Input: 1. dp – dialogue pattern
2.         2. rdfRow - row from RDF results
3.  Output: path – generated path value
4.
5.  path ← create new empty path
6.  longestPP ← find longest pathpattern from ppList of dp
7.
8.  foreach relation pattern rp in longestPP loop
9.      argID1 ← get uri from rdfRow based on the $w_{ra}$
    variable of ap1 in rp
10.     argID2 ← get uri from rdfRow based on the $w_{ra}$
    variable of ap2 in rp
11.
12.     arg1 ← createArgValue(ap1, rdfRow);
13.     arg2 ← createArgValue(ap2, rdfRow);
14.     relation ← new relation with type of rp , source
    argument arg1 and destination argument arg2
15.
16.     add relation to path
17. end loop
```

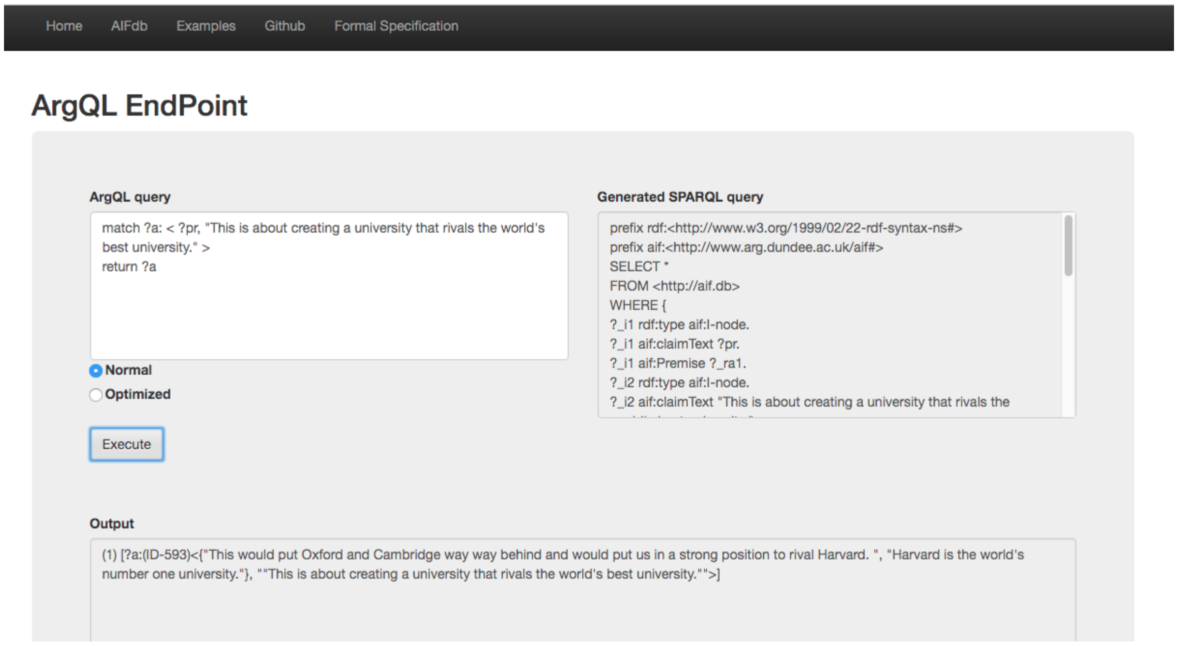
Figure 5.6: Pseudo-code for the transformation between RDF results into argumentative results

the lines 8-16, the longestPP is crossed and for each relation pattern in the sequence, the argument values arg_1 and arg_2 of the included argument patterns ap_1 and ap_2 are created by calling the method *createArgValue* (lines 12-13), which was described above, and also a new relation value with arg_1 and arg_2 being the source and destination arguments (line 14). Finally, the new relation value is added to the returned path value.

5.1.4 ArgQL endpoint

One of the outcomes of this work is a web application that demonstrates an endpoint for ArgQL, where someone is able to test his own queries against the AIFdb dataset. Figure 5.7 illustrates a snapshot of the main page of the application.

The query is written inside the input form with the titular label *ArgQL query*. The query is submitted for execution when pressing the *Execute* button. The text area on the right which has the label *Generated SPARQL query* is an output form in which, after the query has been translated and executed, prints the SPARQL query generated by the suggested translation of section 4.1.2. The results of the ArgQL query are shown in the *Output* area at the bottom of the page. Finally, the set of radio buttons with the choices *Normal* and *Optimized*, are used to set whether the query will be translated based on the optimization suggested in section 4.2, or not.



```
match ?a: < ?pr, "This is about creating a university that rivals the world's
best university." >
return ?a
```

[✓ Normal] Optimized

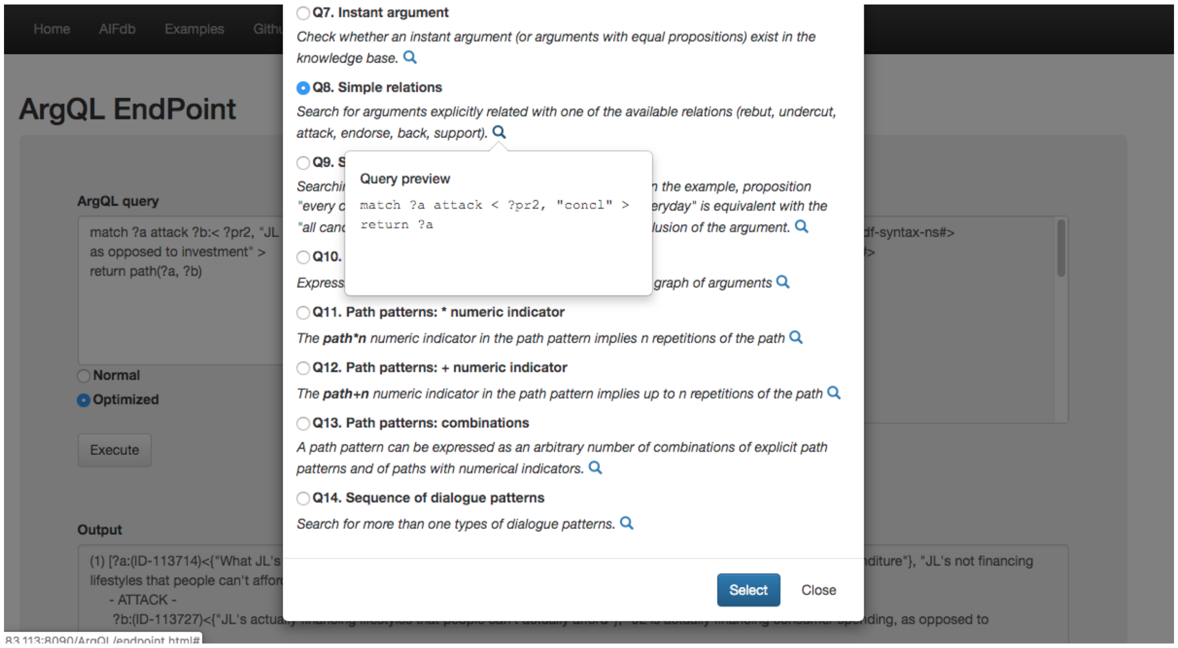
```
prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix aif:<http://www.arg.dundee.ac.uk/aif#>
SELECT *
FROM <http://aif.db>
WHERE {
?_i1 rdf:type aif:I-node.
?_i1 aif:claimText ?pr.
?_i1 aif:Premise ?_ra1.
?_i2 rdf:type aif:I-node.
?_i2 aif:claimText "This is about creating a university that rivals the
```

```
(1) [?a:(ID-593)<{"This would put Oxford and Cambridge way way behind and would put us in a strong position to rival Harvard. ", "Harvard is the world's
number one university."}, ""This is about creating a university that rivals the world's best university."">]
```

Figure 5.7: Snapshot of the ArgQL endpoint

The menu bar on the top of the page, has multiple choices. The *Home* choice is quite trivial and returns to the main page. The *AIFdb* redirects to the link of the AIFdb dataset, and in particular to the ”http://corpora.aifdb.org/”, where someone can find the data in several forms, including their schematic representation. The *Examples* option opens a list of preset queries, which has been created to facilitate the utilization of the application, as well as the familiarization with the syntax of ArgQL. The list captures the whole spectrum of the supported queries . Figure 5.8 shows a snapshot of that list. For each query, there is a description of the informational requirement it expresses and when pressing on the glass icon, a preview of the query is revealed. With the *Select* button, the query at the selected choice of the radio button list is appeared in the *ArgQL query* text area of the main page. The

Gihub choice redirects to the source code of the implementation described in the chapter. Finally, the *Formal Specification* leads to material that contains the formal documentation of ArgQL.



```
match ?a attack ?b:< ?pr2, "JL
as opposed to investment" >
return path(?a, ?b)
```

Optimized

Q7. Instant argument
Check whether an instant argument (or arguments with equal propositions) exist in the knowledge base.

[✓ **Q8. Simple relations**]
Search for arguments explicitly related with one of the available relations (rebut, undercut, attack, endorse, back, support).

Q9. S
Searchi… n the example, proposition "every c… eryday" is equivalent with the "all canc… lusion of the argument.

Query preview

```
match ?a attack < ?pr2, "concl" >
return ?a
```

Q10.
Express… graph of arguments

Q11. Path patterns: * numeric indicator
*The **path*n** numeric indicator in the path pattern implies n repetitions of the path*

Q12. Path patterns: + numeric indicator
*The **path+n** numeric indicator in the path pattern implies up to n repetitions of the path*

Q13. Path patterns: combinations
A path pattern can be expressed as an arbitrary number of combinations of explicit path patterns and of paths with numerical indicators.

Q14. Sequence of dialogue patterns
Search for more than one types of dialogue patterns.

```
(1) [?a:(ID-113714)<{"What JL's … nditure"}, "JL's not financing
lifestyles that people can't affor
    - ATTACK -
    ?b:(ID-113727)<{"JL's actua … nding, as opposed to
```

83.113:8090/ArgQL/endpoint.html#

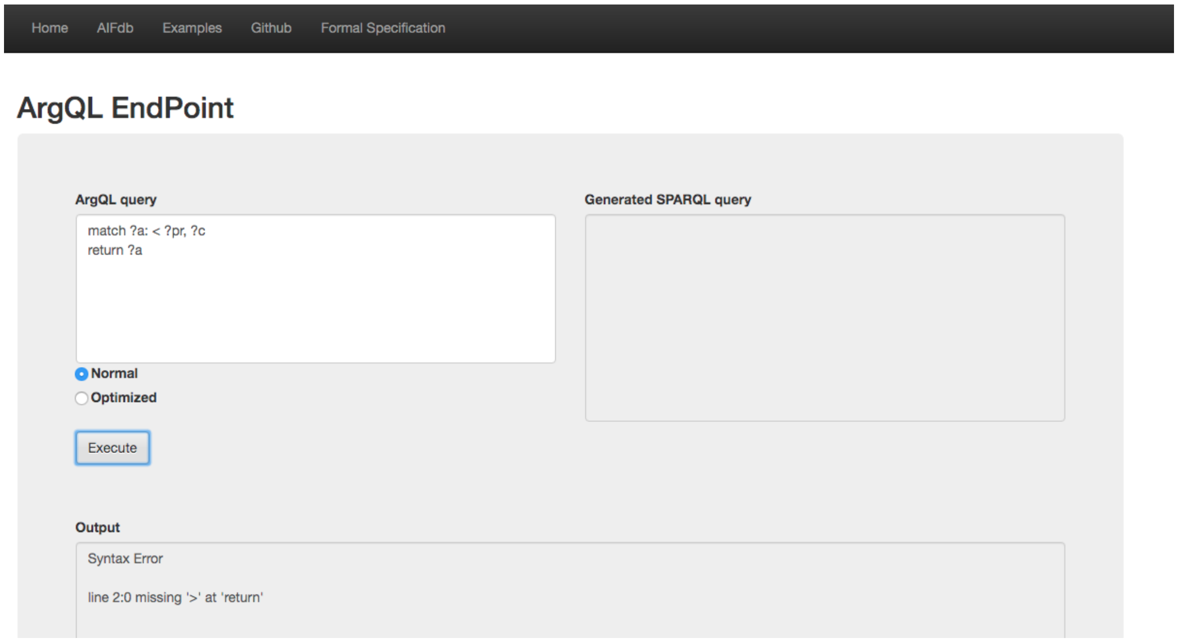
Figure 5.8: Snapshot of the ArgQL endpoint - List of predefined queries

Like any other language, declarative or procedural, ArgQL also supports a mechanism for lexicographic and syntactic validation. When something is not recognized as lexicographically or syntactically correct, the proper messages are appeared on the*Output* form. Fig. 5.9 shows an error example, where the user does not close the argument pattern with th ⟩ character, and the error message printed in the output.

Technologies used

We used several web technologies to implement the ArgQL endpoint. First of all, for the interface (menus, text fields buttons etc.), we used Bootstrap[1], a free and open-source CSS framework, on top of html, which provides a set of pre-designed components to build a polished web page. To manage the responsive behavior of these components (press of buttons, event handing etc) we used jQuery[2], a fast, small, and feature-rich JavaScript library. The methods for query execution and by extension, the translation of the initial query to SPARQL, have been implemented as a RESTful web services api[3]. Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. It is mainly used to develop lightweight, fast, scalable and

[1]https://getbootstrap.com/
[2]https://api.jquery.com/
[3]https://www.tutorialspoint.com/restful/index.htm



```
match ?a: < ?pr, ?c
return ?a
```

```
Syntax Error

line 2:0 missing '>' at 'return'
```

Figure 5.9: Snapshot of the ArgQL endpoint - Error messages

easy to maintain web services, that often use HTTP as a means for communication. Thus, RESTful applications use HTTP requests to post data (create or update), read data (making queries) and delete data. REST is not a standard in itself, but instead is an architectural style that uses standards like HTTTP, XML/HTML/JSON to encode data of the HTTP requests and responses. Finally, we used the Apache Tomcat[4], an open source Web server tool developed by the Apache Software Foundation (ASF), in order to host the web page of the endpoint and the RESTful web services.

[4]http://tomcat.apache.org/

5.2 Evaluation

In this section, we present an extensive experimental evaluation of the suggested methodology for the ArgQL query execution. Our analysis has four main objectives: *(i)* To provide a numerical estimation of the size of the generated SPARQL queries, as well as to show the growth rate of these numbers, when adding different types of patterns to the initial ArgQL query (sec. 5.2.2). We present those numbers for both translation versions: with and without the optimization. *(ii)* To show how the suggested optimization affects the execution times in the different query categories(see list in sec. 5.2.2). *(iii)* To evaluate the extent to which, the addition of a particular type of pattern changes the execution time (sec. 5.2.2). The particular measurement is tested on the optimized translation version. *(iv)* To estimate the scalability of the generated queries (also for the optimized version) in increasing volumes of data (sec. 5.2.2).

5.2.1 Experimental setup

Software

For the RDF data management (e.g. storage of datasets and query execution), we worked with the open source version of the Virtuoso Universal Server[5] (v7.2.4.3217) [42], in which, the RDF triples actually reside in a relational database. On top of that, we used the Java programming language (version 9) to develop the algorithms that realize the translation and we used the API of the Jena open source framework[6] for the communication with Virtuoso. The ArgQL parser is generated (also in Java) from a context-free grammar, using the tool ANTLR v4 [75]. Roughly, ANTLR takes as input a grammar that specifies a language and, adopting the left recursion approach, it generates source code for the recognizer of that language. In respect to the optimization methodology, we maintain a relational database scheme to keep the associations between the RDF literals and the canonical IRIs. To do this, we used MySQL[7], which is an open-source relational database management system (version 8.0.4) and we used the jdbc api [45] to manage the database from our main Java program.

Environment

All experiments were performed on a machine which uses an Intel(R) Core(TM) i7-3770 CPU at 3.40 GHz, with 16.0 GB of RAM,, with a single 7200 RPM 930GB hard drive, running Windows 10 Enterprise, version 1803 (OS build 17134.885). From the total amount of memory, we dedicated 8GB for Virtuoso.

[5]https://virtuoso.openlinksw.com/
[6]https://jena.apache.org/
[7]https://dev.mysql.com

Datasets and Queries

For our experimental evaluation we used the AIFdb dataset[8]. AIFdb is a corpus of argumentative data, gathered from several argumentation tools, such as DGEP [25], the AnalysisWall [23] and ArguBlogging [27]. Table 5.1 gives some indicative sizes that characterize the dataset.

Parameter	AIFdb Sizes
Size on disk	420MB
Number of triples	1.1M
Number of I-nodes	46047
Number of RA-nodes	14208
Number of conflicts(CA-nodes)	13003
Number of default rephrases	2134
Cycles	yes

Table 5.1: Statistical description of AIFdb corpora

The AIFdb dataset consists of 10427 RDF files, that occupy 420MB of disk space. Those files contain data structured in the AIF conceptual scheme. In terms of triples quantity, it includes 1.1 million of them. Some other interesting sizes concern the numbers of instances of some particular concepts in AIF, like the number of I-node instances (46047), the number of RA-node instances (14208), the number of conflict application (CA) instances (13003) and the number of default rephrases (2134). Moreover, the dataset includes cycles, which essentially means that, inside a cycle, the length of a path is infinite. This information is crucial in application domains that deal with the problem of graph traversal. Given that there is a great analogy between the concepts of our argumentation model and the AIF, the particular dataset was inherently suitable to evaluate the applicability of ArgQL beyond the theoretical boundaries.

We used the following 10 ArgQL queries to perform the experiment tests.

Q1. **match** ?a: ⟨ ?pr, ?c ⟩ **return** ?a

Q2. **match** ?a: ⟨ ?pr, ”value” ⟩ **return** ?a

Q3. **match** ?a: ⟨ ?pr[/{”p_1”}], ?c ⟩ **return** ?a

Q4. **match** ?a: ⟨ ?pr[.{”p_1”}], ?c ⟩ **return** ?a

Q5. **match** ?a: ⟨ ?pr_1, ?c_1 ⟩ , ?b: ⟨ ?pr_2[/?pr_1], ?c_2 ⟩ **return** ?a

Q6. **match** ?a: ⟨ ?pr_1, ?c_1 ⟩ , ?b: ⟨ ?pr_2[.?pr_1], ?c_2 ⟩ **return** ?a

Q7. **match** ?a: ⟨ {”p_1”} , ”concl” ⟩ **return** ?a

Q8. **match** ?a *attack* ?b **return** *path(?a, ?b)*

Q9. **match** ?a *attack/attack* ?b **return** *path(?a, ?b)*

Q10. **match** ?a *(attack)+2* ?b **return** *path(?a, ?b)*

[8]http://corpora.aifdb.org/

Q1 is used to evaluate the efficiency in the face of the requirement to return all arguments in the knowledge base. Q2, Q3, Q4 and Q7 are expected to give similar performance results, since they all search for particular arguments, by filtering their internal structure with constant proposition values. Q5 and Q6 are used to evaluate the performance when searching for combinations of arguments, while Q8, Q9 and Q10 are used to evaluate the effectiveness of locating complete paths in the graph.

5.2.2 Results and Discussion

Size of the generated queries

The objective of this measurement is to give an insight about the sizes of the queries, that the translation described in chapter 4 generates. We did this by reducing the query to the graph pattern inside the *where* clause of SPARQL and counting the number of joins and unions contained. The table 5.2 collects those sizes. The characters ’J’ and ’U’ are used for the number of Joins and Unions, respectively. We give the results for both versions of translation, with and without the optimization of the canonical URIs (Normal and Optimized).

		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**
Normal	J	7	22	25	25	16	30	37	94	166	253
	U	0	1	1	1	0	3	2	8	14	24
Optimized	J	7	7	10	10	16	14	7	25	43	68
	U	0	0	0	0	0	0	0	1	2	4

Table 5.2: Sizes of the generated SPARQL in number of JOINS and UNIONS

With a first glance, we draw the (obvious) conclusion that, as the queries become more complex, from Q1 to Q10, the generated graph patterns are getting bigger, for both *normal* and *optimized* case. Q1 is the simplest query and has the smallest generated size. Regarding the Q2-Q4, we observe that they generate queries of similar sizes. We also observe about them, that the number of unions for all is equal to 1. This happens because of the fact that in the particular use cases, they include 1 constant proposition pattern and the union is used to check for equivalent values. Notice also that the queries Q8-Q10, which express the requirement of graph traversal, the numbers are increased significantly. The results in the table also confirm that the queries generated with the optimization, are much smaller than the ones generated without it, in both numbers of joins and joins. Especially for the most complex cases, we observe that the difference is noticeable big. As we will see in the next experiment, that difference has impact on the execution times. The only cases, in which the sizes are equal between the normal and the optimized version, are for the queries Q1 and Q5 and this happens because these are the only cases that are not affected by the equivalence factor.

In the second part of this measurement, we discuss how much, each particular component of the ArgQL specification contributes to the size of the generated query. We concluded that the factors mostly affecting the size are the following five: i) number of proposition patterns ii) number of argument patterns iii) number of dialogue patterns that have the form of a path pattern between two argument patterns iv) length of a path pattern and v) number used in the '+' indicator of a path pattern expression. To conduct this analysis, for each of the five cases, we started with one of the queries in the list above, and by increasing the numbers by one, we count the number of joins and unions of the generated queries, again for both *normal* and *optimized* versions. Certainly, the presented numbers concern those very specific cases and a query could contain arbitrary combinations of them, that would give completely different numbers. However, the aim here is to give a general intuition about which components are the ones that contribute more to the size. The results of this analysis are shown in figure 5.10. The five charts depicted, correspond to the five cases discussed before. Regarding the abbreviations used, *JN* stands for the number of joins in the normal case, *UN* stands for the number of unions in the normal case, *JO* is the number of joins in the optimized case and *UO* stands for the number of unions in the optimized case.

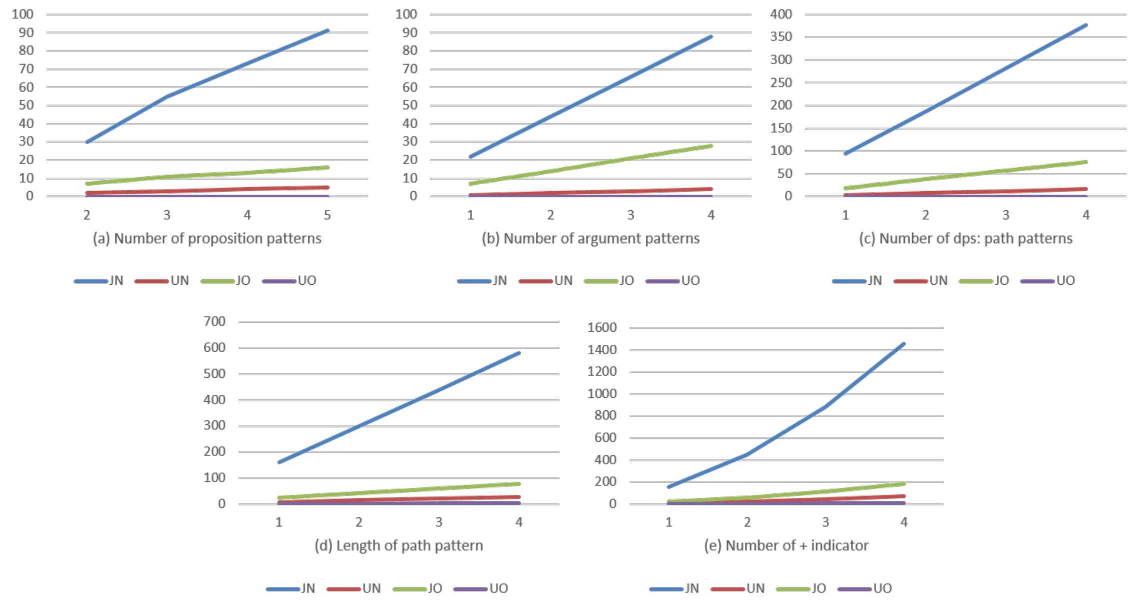


Figure 5.10: Estimation about how the size of the SPARQL query is affected when adding constructs in the ArgQL query

About the first case (number of proposition patterns) the results are shown in diagram (a). We start with the query Q7 and gradually add proposition patterns in the premises of the argument pattern. We notice a linear and steady increment for all of the four sizes and this is normal, since, independently of the proposition pattern, the graph pattern to which it corresponds has fixed size in both normal and optimized versions. In the normal version, each new proposition pattern contributes with 18 joins and 1 union, while in the optimized case it only contributes with 3 joins and zero unions. This is the reason why the blue line *(JN)* lies so much higher than the other three. Note also that a query with only five proposition patterns can generate a SPARQL with 90 joins in the normal case.

About the second case (number of argument patterns) the results are shown in diagram (b). We start with the query Q2 and gradually add argument patterns of the same form. Thus, the linear and steady increment of the 4 sizes is also normal, here. In the normal version, each new argument pattern contributes with 22 joins and 1 union (since it contains a proposition pattern), while in the optimized version, it contributes with 7 joins and zero unions. A query with four argument patterns can also generate a SPARQL so elongated as 90 joins.

About the third case (number of path patterns) the results are shown in diagram (c). We start with the query Q8 and gradually increase the number of dialogue patterns by one of the same type (path pattern containing a single *attack* relation pattern). Thus, the linear and steady increment of the 4 sizes is also normal, here. In the normal version, each new such path pattern contributes with 94 joins and 4 unions, while in the optimized version, it contributes with 19 joins and zero unions.

About the fourth case (length of path patterns) the results are shown in diagram (d). We start with the query Q8 and gradually increase the length of the path pattern by one each time which is of the same type. We also notice here a linear increment in the 4 sizes and this is because each new relation pattern in the sequence of the path pattern generates a fixed sized graph pattern (one new intermediate argument pattern and the translation of the relation pattern). In particular, in the normal version, the size is increased by 140 joins and 7 unions, because for each attack relation pattern, we check all the possible cases between the two conclusion patterns (in case of rebut) and all the possible cases between the conclusion pattern of the first and the premise pattern of the second (in case of undercut). In the optimized version, it contributes with 18 joins and 1 union (only for the check between *rebut* and *undercut*).

About the fifth case (number in the '+' indicator), the results are shown in diagram (e). Again here, in the query Q10 we start with the number 1 (equal to Q8) and we increase it by one each time. In this case, the increment is linear but it is not steady and this is because, as the number increases, the size of the generated graph pattern becomes longer. Roughly, if x is the length of the graph pattern of the included relation pattern r, then the length of the expression *(r)+n* will be given by the following:

$$l = \sum_{i=0}^{n} ix$$

The l here refers to both *JN* and *JO* sizes. This case is the most costly in the sense that it causes the longest queries among the five factors. Note that for a simple query that uses the particular path pattern expression with the number 4, the generated query in the normal case exceeds the 1400 number of joins and the 70 number of unions, while in the optimized case, those numbers are 187 and 13, respectively. Thus, the outcome of this case constitutes one more tactile argument in favor of the optimized translation.

Optimization efficiency

General Remark. In the following experiments, we evaluate the execution time of the queries. That time is essentially the total of the individual times of all the steps in the execution cycle: 1) the time to parse and translate to SPARQL, 2) the execution time of SPARQL in virtuoso and 3) the time to collect RDF data and transform them to argumentative form. Each experiment was repeated ten times with the best and worst results discarded and the remaining eight numbers averaged. In all of our experiments, the results were very similar among these eight runs and thus, the confidence intervals were too small.

In this part of our experimental analysis, we compare the execution time between the initial translation and the optimized one. In particular, we examine in which cases the suggested optimization succeeds better, similar, or worse execution time, and how much exactly is that time. The outcomes of the experiment are presented in figure 5.11. For each query, we give 4 numbers. With the blue and the green columns *(normal-exec, optimized-exec)*, we represent the execution time of the SPARQL query, (without the parsing and collect time) for the normal and optimized approaches respectively, while with the red and purple columns *(normal-total, optimized-total)*, we give the total execution time until the results are returned to the ArgQL query. We made this separation so that we can construe and understand better which steps affect the execution time more and detect the bottlenecks within the execution cycle for each case.

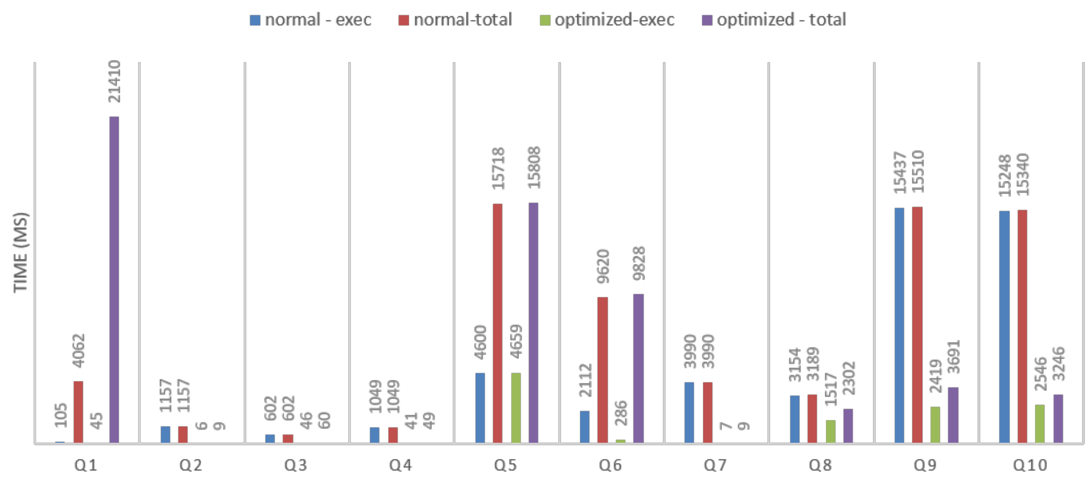


Figure 5.11: Effect of the optimization in the execution time

About Q1, we observe similar execution times of the generated query between the two cases, but a significant difference in the total times, with the optimized one to be greater. This is the only case in which the optimized version is slower and this happens because of the great number of results (all arguments in the knowledge base) and because, at collect time, the values for all of the propositions in the results must be retrieved from the database used to store the information for the canonical URIs. The next three queries Q2-Q4 have all similar results. In particular, for all of them, the optimization succeeds much better execution and total time, but generally those queries are fast in both cases. The query Q5, as we mentioned also in the previous section, does not take into account the equivalence factor, thus it was expected that they would give similar results in the two cases. We notice that this query is executed in about 4,5 seconds but it gives a great volume of results, which at collect time was too expensive to process. For that reason, we used the *limit* option of SPARQL, and we reduced the number of results to 200, such that we would be able to manage them. Even with that limitation, we observe that it takes about 15 secs to return the results to the ArgQL query. Note that in this query, the double negation *(NOT EXISTS)*, caused a great delay in the execution of the particular query. Therefore, we replaced one of the two *NOT EXIST* filters with the *NOT IN* and we succeeded a satisfying execution time. The next query (Q6), which expresses the requirement of join between the premises of two arguments, has a bit faster execution time in the optimized version, but the total times are almost the same. A significant improvement is observed in the execution time of Q7. This is because that query highly depends on the equivalence factor which means that, the more proposition patterns appear in the particular query, the greater the difference between the two versions will be. A considerable improvement in the optimized approach is observed also for the remaining queries Q8-Q10 and generally for queries that search for complete paths in the graph. Note that for all those queries we ask for any path in the knowledge base, that confronts to the referred path pattern. Given also the existence of cycles, the SPARQL query returns a huge volume of data which is also for this case, difficult to process and transform at collect time. Therefore, we also used here the expression *limit 1000* to restrict them. As a result we see that without the optimization, the total time was ~ 3 seconds for Q8 and ~ 15 seconds for Q9 and Q10, whilst with the optimization, the total time is a bit smaller for Q8 (~ 2 seconds) and much smaller for Q9 and Q10 (~ 3 seconds).

Overall, we conclude that in most of the cases, the optimization improves significantly the execution time, especially for the queries that use constant propositions to filter the structure of argument and for those that search for relations or sequences of relations between arguments. However, when the queries must return a great number of results, the extra step of retrieving the text values from the database of canonical elements during the results transformation, causes a small delay in the process. That issue was resolved by using the *limit* option of the SPARQL such that the results are retrieved and processed in portions.

Time efficiency

In this section of our experimental analysis, we assess how the various components in the ArgQL syntax affect the execution time. In particular, we focus on the factors described in the first experiment of the evaluation. Therefore, we are going to discuss the temporal burden in the following cases: 1) when a new proposition pattern is added 2) when a new dialogue pattern is added (single argument pattern or a path pattern), 3) when the length of a path pattern is increased and 4) when the number *n* in the path pattern expression *(r)+n* is increased. The aim of this experiment is to give an idea about the restrictions of the particular implementation of ArgQL, against its syntax. The outcomes are presented in figure 5.12. The translation version we evaluate here is the optimized one. All of the following experiments concern the total execution time (including the time for parsing, translation, execution and results collection). For the current experiment, we set the threshold for the response time, between *20-30* seconds. The waiting time above that threshold can be considered as non tolerable. As a result, we increased the numbers of the referred factors, until the execution time reaches that threshold and we discuss for each case the rate of increment.

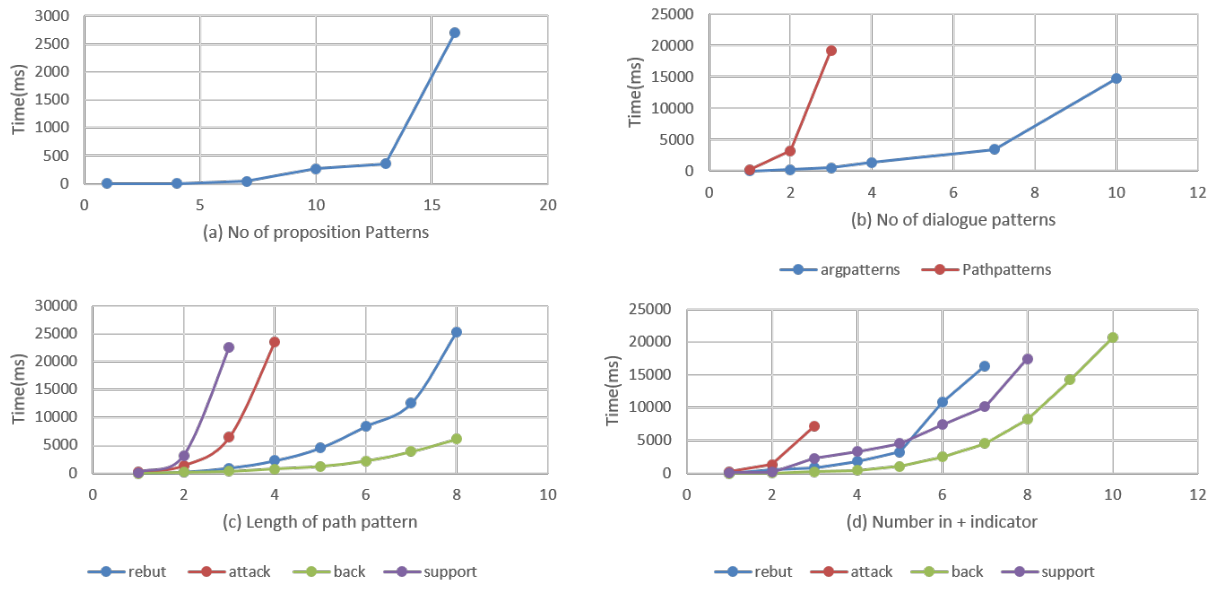


Figure 5.12: Estimation about how the execution time of the SPARQL query is affected, when adding constructs in the ArgQL query

To create the first diagram (a) we start with the query Q2, that contains one proposition pattern in the conclusion part and then, we gradually add proposition patterns in the premise part. We observe in the diagram that, for a number of proposition patterns greater than 12 in the same query, the execution time gets increasingly big, while for more than 16 proposition patterns, it exceeds the predefined threshold.

In the second chart (b) we merge the cases where a dp is a single argument pattern with the case where it includes a path pattern, in the same diagram. We observe that the maximum number of path patterns a query can ”tolerate” is three, while the maximum number of single argument patterns is 10. Above those numbers, the queries exceeded the temporal threshold.

The third chart (c) presents the execution time for different lengths of paths and of different types depending on the relation patterns existing in the sequence. Here, we present four of the six supported relations: rebut, attack, back and support. The other two (undercut, endorse) have very similar results with the rebut and back respectively and therefore they were omitted. In general, we notice that longer path patterns (length of eight or more) of the rebut and back relation type (and by extension undercut and endorse) can be supported, while for paths patterns that consist of the abstract relations support and attack, we can only have path pattern of length three and four, respectively. This was expected since in the abstract relations, more cases need to be checked and as we showed in the first experiment, the size of the query is greater for these cases. As a result, if there is a need to express long path patterns, we recommend using the more specific types of relations and avoid the most general ones.

The fourth chart (d) shows the changes in the execution time when the number *n* increases in the *(r)+n* path pattern expression. We examine the execution times for all of the different types that *r* may get (again here the undercut and endorse are omitted for the same reasons as before). We see that regarding the *attack* case, the maximum number we can use is three, (*(attack)+3*) while for the *back* relation, that number can be even 10 (ten alternative sequences of *back* type can be supported). This is easily explained, since in the optimized version that we examine, the back relation is expressed with just an equality check between the canonical values that correspond to the involved conclusions.

Scalability

One of the most crucial issues when processing data, is the extent to which the proposed methodologies scale in increasing volumes of data. This is what we investigate in this set of experiments: how efficient ArgQL remains, when the sizes of the target data increases. To realize the analysis, we started with the initial dataset that consisted of ~ 1.1M tuples according to the table 5.1 and then we multiplied this by 5 and by 10, creating that way two new datasets of ~ 5.5M and ~ 11M tuples, respectively. For each of the three datasets, we ran the queries Q1-Q10 and measured the execution time in the way that we described above (averaged of eight execution times). Figure 5.13 reports our results.

At a first glance, we notice a negligible increment to the execution times across the three data sets for most of the queries. In particular, the analysis showed that queries Q1, Q2, Q3, Q4, Q6, Q7 and Q8 scale well in the particular sizes that we used, creating the prospect of similar scalability results, for even larger data sets. On the other hand, queries Q5, Q9 and Q10 seem not to be so scalable. Regarding Q5, we already discussed the reasons why it exhibits such a high execution time and this is because of the double negation, which has a great impact to the performance especially in big volumes of data. Thus, we observe that in the largest data set, a query of this type, may take even one minute to be executed.

About the queries Q9 and Q10, we discern two cases for each one. In the first case, the queries Q9 and Q10 are translated as ”search for *any* path matching to the particular pattern” and this is expressed with the use of single variables as argument patterns, which, according to the semantics, match any argument in the knowledge base. In the second case, we assume two new queries Q9’ and Q10’, which are of the same type as Q9 and Q10, but differ in that, at least one of the two argument patterns are restricted to locate particular arguments. For example, they express queries like ”search for paths that match to the defined path pattern, terminating to (or beginning with) arguments with a particular conclusion.” We observe that in the first case, the queries Q9 and Q10 are not that scalable. In particular Q9 takes about 2.5 minutes to be executed in the largest dataset, while Q10 fails to

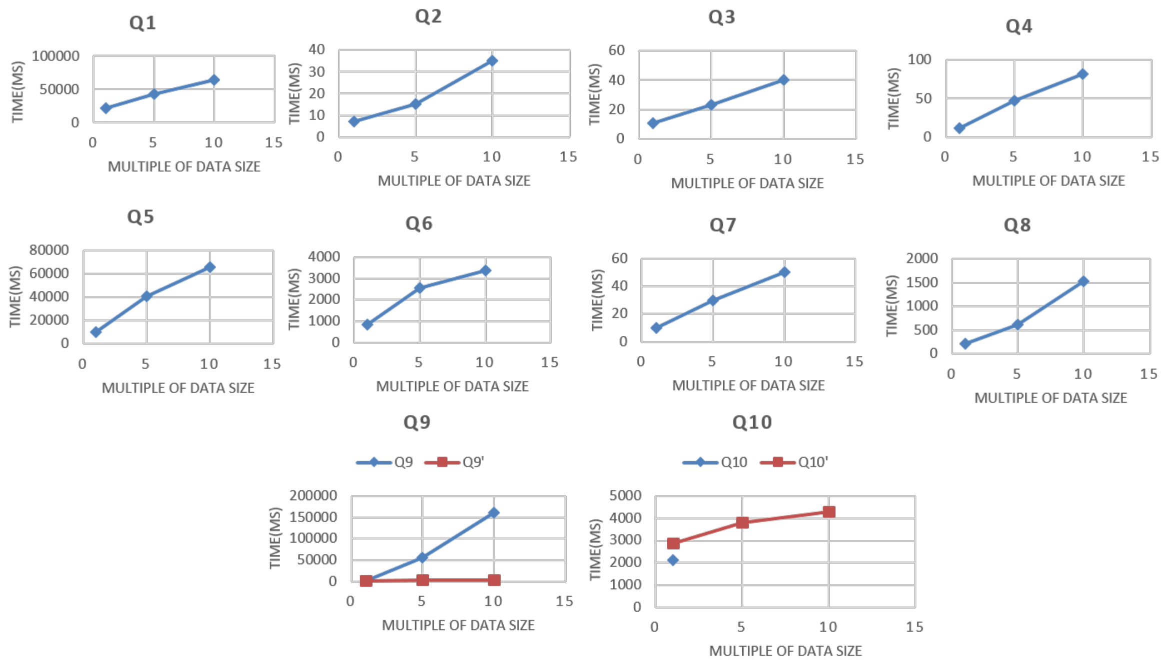


Figure 5.13: Scalability in data size

be executed even in the medium data set. The explanation for this delay, lies in the large number of data that the SPARQL engine has to combine in order to generate the answer (wide number of variables and many result rows). On the contrary, queries Q9' and Q10' seem to be quite scalable and in particular, Q9' takes about 2.5 seconds to be executed in the largest dataset, while Q10' needs about 4.5 seconds. This happens because of the small number of result rows of SPARQL having as consequence the query to return the answer fast.

Overall, we can say that the particular implementation of ArgQL to the specific dataset, scales well for most of the simple cases. The only cases in which the results are not that satisfying, are when we search for long paths without doing any further restriction on this search. For these cases, it is recommended to restrict the required matching paths in one of the source or destination argument patterns. Of course, as the queries become more complex, it is inevitable that the scalability will be affected, as well. In the future we may need to improve more our implementation, so that any problematic case revealed by this experiment to improve in performance in large volumes of data.

Note also that the two new datasets are just copies of the initial one. This means that, although the size is multiplied, the topology of the generated data remains the same. By topology, we mean the structure of the data, the length of the existing paths etc. However, the existence of cycles in the graph of arguments, allowed us to test paths patterns of various lengths, and that way we were exempted by the need to create synthetic data in order to test our queries in data of different topology.

Chapter 6
Conclusions and Future Directions

Contents

We conclude this thesis by summarizing its main contributions and by making a discussion about the research directions in which this work can be continued.

6.1 Synopsis of Contributions

In this thesis, we were incited by the challenge to study and formalize the process of information searching from structured dialogues. Based on the claim that the problem is associated with a set of very specific and concrete information requirements, we were impelled to isolate and bring them together in the formal specification of a query language. That motive lead us to ArgQL, a query language which, as far as we concern, constitutes the first language designed for this purpose. The particular aim of ArgQL, was to provide a simple and quite elegant machinery to search for pieces of data within collections of structured dialogues, by offering pure syntax and built-in, dialogue-related terminology.

At first, we recorded the types of queries supported by ArgQL and the set of information requirements they would cover. The initial version of the language, presented in this document, captures a subset of the requirements described in the motivation section. In particular, we cover 4 general categories of queries: *a) Locating individual arguments b) Identifying commonalities in argument's structure, c) Extracting argument relations and d) Traversing the argument graph.* Of course, even in these categories, the capabilities offered by ArgQL are not exhaustive. Instead, all of them can be extended with more functionality in later versions of the language. We will talk about possible extensions in the next section.

Subsequently, we defined the theoretic principles that surround the data model, according to which the target data are structured. To do this, we addressed to the formal models in the area of Computational Argumentation, since this area, for many years studies the way of human reasoning while arguing and its computational models are based on representations which are rich with semantics in the domain. We designed our data model to be as general as possible, so that it would be compatible with the majority of all these models and this was verified by the latter comparison with the AIF conceptualization, an ontology designed exactly for this reason: to represent as more as possible of the semantics of the different argumentation models.

In chapter 3 we gave the formal definition of ArgQL and its main constructs. In particular, we designed ArgQL as a declarative language, the functionality of which is based on the idea of pattern matching. Its formal specification includes a set of patterns, which range from simple ones, to more complex, composed by others. After having defined formally the different types of patterns, we present the syntax of ArgQL as an EBNF grammar. We closed the chapter of the formal specification, by defining the semantics of the language. ArgQL semantics are based on an interpretation function that describes, each different type of pattern, to which sets of elements from the data model will match. This function behaves recursively as the patterns are composed from the simpler, to the most complex ones.

In the remainder of this thesis, we dealt with issues that regard the implementation of ArgQL. In particular, we adopted the approach of translating the language to some of the standard query languages and take advantage of their efficiency, as well as of existing data-sets stored in the respective format. We show the case of RDF/SPARQL, since it is the standard representation and query languages in Semantic Web, which is also our domain of interest. Regarding the mapping of the data models, we chose the AIF ontology to create the RDF data and we showed each element of our data model, to which concepts of AIF correspond. As for the query translation, we broke the process in a set of rules, such that, each rule describes the translation of a particular type of pattern. The soundness and completeness of the process is also formally proven. Given that the proposed translation generated non-optimal queries, we also suggested an optimization, which was essentially based on the elimination of the equivalence factor from anywhere appearing in the query. In the majority of the queries, the optimization indeed succeeded better performance and this was confirmed in the experimental evaluation that followed.

Finally, one equally significant outcome of this work, was the ArgQL endpoint, which gives the opportunity for someone to test ArgQL queries in real data-sets and familiarize himself with the language.

6.2 Directions for Future Work and Research

ArgQL featured new aspects and challenges in the problem of information seeking in argumentative dialogues and through this new research mantle, a number of problems are emitted that still remain open and would be worthy to investigate. In this section we outline some directions for the continuation of this work.

6.2.1 Extensions in the Language

Although ArgQL already covers a wide variety of information needs related to the domain, there still exist interesting things to ask from a dialogue, which are not supported by its current version. We propose here some possible extensions.

Meta-data and keyword searching enrichment

Apart from its main informational content, a dialogue is also characterized by its meta-information, that corresponds to the context in which it was created. We refer to information like its main topic or sub-topics, information about who expressed an opinion (provenance, trustworthiness etc), temporal aspects like when it was expressed and with which events it concurred etc. An interesting extension would be to enrich the ArgQL syntax such that it will support also the querying of such information. For example someone will be able to ask for arguments by a particular person, or of a specific topic or subtopic, or even query about trustworthiness of the resource. In such a case, the semantics and the implementation would have to be adjusted to these new features, as well. This extension, although straightforward, would highly strengthen the usefulness of the language and would broaden its expressive scope to more meaningful and realistic searching use cases.

The practicality of the current version of ArgQL is a bit handicapped by a particular factor: the constant proposition values. In real scenarios, it would be almost impossible for the query writer to know exactly the propositions, with which he wants to make the search. Since our target scenario is the web and such cases demand practical solutions, one more idea in order to tackle that issue, is to incorporate facilities that allow ”smart” searching 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. The requirements about implementing this extension would be mainly focused on including the appropriate syntax and of course, to add it in the implementation.

Querying about argument validity

One of the most valuable extensions to investigate is the integration of mechanisms for computing the validity of the arguments and also enable for querying about it. In the area of Computational Argumentation, there is a variety of models that deal with the evaluation of the acceptability and the persuasiveness of arguments and it would be quite interesting to examine the case of incorporating some of them in the specification of ArgQL.

The graph view of our data, allows us to concern the semantics of Dung [38] for this feature and support queries about the acceptability of arguments. The different types of extensions could also be taken into account during this process. That feature, combined with the rest of the functionality supported so far, would give the opportunity to express hybrid queries like ”Is an argument with a particular conclusion accepted under the extension X?” or ”Find the acceptable arguments which have common premises with a particular argument” etc.

Apart from the semantics of Dung though, there are many more models that compute numerically the acceptance of arguments. For example argumentation systems that use weighted arguments (e.g. [5–7, 40]) or weighted attacks (e.g. [36, 41]) can offer useful algorithms to incorporate in ArgQL. Queries of this category would be like: ”Find arguments with acceptability degree greater than 0.7” etc. The notion of persuasiveness (or impact) of an argument includes the parameter of the audience. A potential extension would be of this kind. In particular, inluding methods of argumentation frameworks that evaluate the persuasiveness of arguments from the audience’s point of view [16, 55, 58, 59, 83], could also give interesting queries like: ”Find the most convincing arguments for an audience with particular profile characteristics”.

The feature for acceptability is associated with various sub-problems. Giving the opportunity for the query writer to determine the scope in which it will be computed and in particular to decide whether it will perform in a sub-graph given in the query, or if it will happen in the entire data-set each time is one of them. Another decision regards the time in which that computation would be performed. For example whether acceptance would be computed once in periodical intervals and would be kept offline as part of the information in the data model, or whether it would be computed on the fly, at query time and whenever asked.

Ranking and Grouping results

A part within the specification of ArgQL which can also accommodate extensions is the way in which the results are returned.

The significance of the language would be intensified if we provided mechanisms to rank the returning results. For example if giving the opportunity to determine the ranking method by choosing among a set of available options like : strength of arguments, creation date, trustworthiness of the resource, relevance to the audience etc. Ideas for this feature can also be found in the area of strategic argumentation [60, 110, 113], the studies of which focus on selecting the best argument to communicate, based on the standing conditions and the state of the dialogue.

Finally, like most of the common query languages, we could provide the capability of grouping the returning results, again according to specific criteria, like resource, date, su-topic etc.

Argument Relevance

One more useful idea is to extend ArgQL with the capability to search for relevant arguments. For example given a specific argument, ask for arguments with a particular degree of relevance. There are many different strands to the concept of relevance. It can be computed based on the commonalities in the arguments’ structure, (e.g. two arguments with a number of common (or equivalent) premises, or arguments being mutually supporting, could be considered as relevant), but it could also be computed according to the topics and sup-topics they touch upon.

To some extent, the current version of ArgQL supports this feature via the *inclusion and jointness* filters of the premise pattern. However they represent a kind of simplified concept of relevance. In case we want to seriously incorporate this feature in the language we should define a model for relevance in the data model (probably based on works like [54,63,118]) and then extend the language to ask for relevant arguments, based on this model.

6.2.2 Implementation

Finally, we suggest some directions for future work that concern the implementation of ArgQL. First of all, the proposed implementation can be further optimized, in order to tackle limitations revealed at the experimental evaluation, like supporting longer paths, increasing the number of dialogue patterns in the same query, improving the scalability of the language in the cases that it was not that scalable etc. Furthermore, in order to take advantage of other existing data-sets but most importantly to reach closer to our initial vision of ArgQL being a generic query language for argumentation, it is necessary to translate it to more query languages (like SQL, Cypher, etc) and provide complete libraries that will include all of these mappings. It would also be useful to conduct an experimental evaluation with real users, in order to estimate the easiness for expressing a query in ArgQL compared to other query languages.

Bibliography

[1] Discoursedb: The user-powered database of political commentary.

[2] Araucaria: Software for Puzzles in Argument Diagramming and XML. Technical report, Department of Applied Computing. University of Dundee Technical Report., 2001.

[3] Convinceme, 2012.

[4] Leila Amgoud, Sihem Belabbès, and Henri Prade. A formal general setting for dialogue protocols. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 13–23. Springer, 2006.

[5] Leila Amgoud and Jonathan Ben-Naim. Axiomatic foundations of acceptability semantics. In *Fifteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2016.

[6] Leila Amgoud and Jonathan Ben-Naim. Evaluation of arguments in weighted bipolar graphs. *International Journal of Approximate Reasoning*, 99:39–55, 2018.

[7] Leila Amgoud, Jonathan Ben-Naim, Dragan Doder, and Srdjan Vesic. Acceptability semantics for weighted argumentation frameworks. In *IJCAI*, volume 2017, pages 56–62, 2017.

[8] Leila Amgoud and Claudette Cayrol. On the acceptability of arguments in preference-based argumentation. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pages 1–7, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[9] Leila Amgoud and Florence Dupin De Saint Cyr. Measures for persuasion dialogs: A preliminary investigation. *Frontiers in Artificial Intelligence and Applications*, 172:13, 2008.

[10] Susan Armstrong, Alexander Clark, Er Clark L, Maria Georgescul, Vincenzo Pallotta, Andrei Popescu-behs, Martin Rajman, Maria Georgescul, Marianne Starlander, Ereti. Unige. Ch, Giovanni Coray, Giovanni Coray, Vincenzo. Pallotta Epfl. Ch, David Portabella, and David Portabella. Natural language queries on natural language data: a database of meeting dialogues, 2003.

[11] Katie Atkinson, Pietro Baroni, Massimiliano Giacomin, Anthony Hunter, Henry Prakken, Chris Reed, Guillermo Simari, Matthias Thimm, and Serena Villata. Towards artificial argumentation. *AI magazine*, 38(3):25–36, 2017.

[12] Katie Atkinson, Trevor Bench-capon, and Peter Mcburney. Computational representation of practical argument. *Synthese*, 152:2006, 2005.

[13] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. Review: An introduction to argumentation semantics. *Knowl. Eng. Rev.*, 26(4):365–410, December 2011.

[14] Amine Bayoudhi, Hatem Ghorbel, and Lamia Hadrich Belguith. Question answering system for dialogues: A new taxonomy of opinion questions. In *Proceedings of the 10th International Conference on Flexible Query Answering Systems - Volume 8132*, FQAS 2013, pages 67–78, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[15] T. J. M. Bench-Capon. Deep models, normative reasoning and legal expert systems. In *Proceedings of the 2Nd International Conference on Artificial Intelligence and Law*, ICAIL '89, pages 37–45, New York, NY, USA, 1989. ACM.

[16] Trevor JM Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *Journal of Logic and Computation*, 13(3):429–448, 2003.

[17] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[18] Philippe Besnard and Anthony Hunter. Towards a logic-based theory of argumentation. In *In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'2000*, pages 411–416. MIT Press, 2000.

[19] Philippe Besnard and Anthony Hunter. A logic-based theory of deductive arguments, 2001.

[20] Philippe Besnard and Anthony Hunter. *Elements of Argumentation*. The MIT Press, 2008.

[21] Floris Bex, Thomas Gordon, John Lawrence, and Chris Reed. *Interchanging arguments between Carneades and AIF*, pages 390–397. Frontiers in artificial intelligence and applications. IOS Press, 2012.

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

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

[24] 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.

[25] Floris Bex and Chris Reed. Dialogue templates for automatic argument processing. In *COMMA*, pages 366–377, 2012.

[26] Floris Bex and Christopher Reed. Schemes of inference, conflict, and preference in a computational model of argument. *Studies in Logic, Grammar and Rhetoric*, 23(36):39–58, 2011.

[27] Floris Bex, Mark Snaith, John Lawrence, and Chris Reed. Argublogging: An application for the argument web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25:9–15, 2014.

[28] Elizabeth Black and Anthony Hunter. An inquiry dialogue system. *Autonomous Agents and Multi-Agent Systems*, 19(2):173–209, 2009.

[29] Katarzyna Budzyńska and Magdalena Kacprzak. A logic for reasoning about persuasion. *Fundamenta Informaticae*, 85(1-4):51–65, 2008.

[30] Elena Cabrio and Serena Villata. Five years of argument mining: a data-driven analysis. In *IJCAI*, pages 5427–5433, 2018.

[31] Dan Cartwright and Katie Atkinson. Political engagement through tools for argumentation. In *Proceedings of the 2008 Conference on Computational Models of Argument: Proceedings of COMMA 2008*, pages 116–127, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[32] Dan Cartwright and Katie Atkinson. Using computational argumentation to support e-participation. *Intelligent Systems, IEEE*, 24(5):42–52, 2009.

[33] C. Cayrol and M. C. Lagasquie-Schiex. On the acceptability of arguments in bipolar argumentation frameworks. In Lluís Godo, editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 378–389, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[34] 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.

[35] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537, 2011.

[36] Sylvie Coste-Marquis, Sébastien Konieczny, Pierre Marquis, and Mohand Akli Ouali. Weighted attacks in argumentation frameworks. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.

[37] Brickley Dan and Guha R. V. Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/rdf-primer/, 2004.

[38] 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.

[39] Phan Minh Dung, Robert A Kowalski, and Francesca Toni. Assumption-based argumentation. In *Argumentation in Artificial Intelligence*, pages 199–218. Springer, 2009.

[40] Paul E Dunne, Anthony Hunter, Peter McBurney, Simon Parsons, and Michael Wooldridge. Inconsistency tolerance in weighted argument systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 851–858. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[41] Paul E Dunne, Anthony Hunter, Peter McBurney, Simon Parsons, and Michael Wooldridge. Weighted argument systems: Basic definitions, algorithms, and complexity results. *Artificial Intelligence*, 175(2):457–486, 2011.

[42] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[43] Valentinos Evripidou and Francesca Toni. Quaestio-it.com: a social intelligent debating platform. *Journal of Decision Systems*, 23(3):333–349, 2014.

[44] Xiuyi Fan and Francesca Toni. Assumption-based argumentation dialogues. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[45] Maydene Fisher, Jon Ellis, and Jonathan C. Bruce. *JDBC API Tutorial and Reference*. Pearson Education, 3 edition, 2003.

[46] UC Berkeley's Center for NewMedia. Opinion space.

[47] John Fox, Paul Krause, and Morten EIvang-Gøransson. Argumentation as a general framework for uncertain reasoning. In *Uncertainty in Artificial Intelligence, 1993*, pages 428–434. Elsevier, 1993.

[48] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

[49] Manola Frank and Miller Eric. RDF primer. http://www.w3.org/TR/rdf-primer/, 2004.

[50] K. Freeman, University of Oregon. Dept. of Computer, Information Science, and A. Farley. *Toward Formalizing Dialectical Argumentation*. Number no. 13 in CIS-TR / Department of Computer and Information Science, University of Oregon. Department of Computer and Information Science, University of Oregon, 1993.

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

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

[53] Thomas F Gordon, Henry Prakken, and Douglas Walton. The carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10):875–896, 2007.

[54] Diana Grooters and Henry Prakken. Two aspects of relevance in structured argumentation: Minimality and paraconsistency. *Journal of Artificial Intelligence Research*, 56:197–245, 2016.

[55] Davide Grossi and Wiebe Van Der Hoek. Audience-based uncertainty in abstract argument games. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[56] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5-6):907–928, 1995.

[57] Wael Hamdan, Rady Khazem, Ghaida Rebdawi, Madalina Croitoru, Alain Gutierrez, and Patrice Buche. On ontological expressivity and modelling argumentation schemes using cogui. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 5–18. Springer, 2014.

[58] Anthony Hunter. Making argumentation more believable. In *AAAI*, volume 4, pages 269–274, 2004.

[59] Anthony Hunter. Towards higher impact argumentation. In *AAAI*, pages 275–280, 2004.

[60] Anthony Hunter. Modelling uncertainty in persuasion. In *International Conference on Scalable Uncertainty Management*, pages 57–70. Springer, 2013.

[61] Anthony Hunter. A probabilistic approach to modelling uncertain logical arguments. *International Journal of Approximate Reasoning*, 54(1):47–81, 2013.

[62] H. Jakobovits and D. Vermeir. Robust semantics for argumentation frameworks. *Journal of Logic and Computation*, 9:215–261, 1999.

[63] William Jiménez-Leal and Christian Gaviria. Similarity, causality and argumentation. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 35, 2013.

[64] Paul Krause, Simon Ambler, Morten Elvang-Goransson, and John Fox. A logic of argumentation for reasoning under uncertainty. *Computational Intelligence*, 11(1):113–131, 1995.

[65] Werner Kunz and Horst WJ Rittel. *Issues as elements of information systems*, volume 131. Citeseer, 1970.

[66] John Lawrence, Floris Bex, Chris Reed, and Mark Snaith. Aifdb: Infrastructure for the argument web. In *COMMA*, pages 515–516, 2012.

[67] John Lawrence and Chris Reed. *AIFdb Corpora*, pages 465–466. Frontiers in artificial intelligence and applications. IOS Press, 2014.

[68] John Lawrence and Chris Reed. Argument mining using argumentation scheme structures. In *COMMA*, pages 379–390, 2016.

[69] Marco Lippi and Paolo Torroni. Argument mining: A machine learning perspective. In *International Workshop on Theory and Applications of Formal Argumentation*, pages 163–176. Springer, 2015.

[70] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[71] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[72] Sanjay Modgil. Reasoning about preferences in argumentation frameworks. *Artificial Intelligence*, 173(9):901–934, 2009.

[73] Jann Müller and Anthony Hunter. Deepflow: Using argument schemes to query relational databases. In *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, pages 469–470, 2014.

[74] University of Liverpool. Parmenides.

[75] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[76] Simon Parsons, Katie Atkinson, Karen Zita Haigh, Karl N Levitt, Peter McBurney, Jeff Rowe, Munindar P Singh, and Elizabeth Sklar. Argument schemes for reasoning about trust. *COMMA*, 245:430, 2012.

[77] Simon Parsons and Anthony Hunter. A review of uncertainty handling formalisms. In *Applications of uncertainty formalisms*, pages 8–37. Springer, 1998.

[78] Simon Parsons, Carles Sierra, and Nick Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and computation*, 8(3):261–292, 1998.

[79] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[80] John L. Pollock. Defeasible reasoning. *Cognitive Science*, 11:481–518, 1987.

[81] Martin Potthast, Lukas Gienapp, Florian Euchner, Nick Heilenkötter, Nico Weidmann, Henning Wachsmuth, Benno Stein, and Matthias Hagen. Argument search: Assessing argument relevance. 2019.

[82] Henry Prakken. Coherence and flexibility in dialogue games for argumentation. *Journal of logic and computation*, 15(6):1009–1040, 2005.

[83] Henry Prakken. Formal systems for persuasion dialogue. *The knowledge engineering review*, 21(2):163–188, 2006.

[84] Henry Prakken. An abstract framework for argumentation with structured arguments. In *IN INFORMATION TECHNOLOGY AND LAWYERS: ADVANCED TECHNOLOGY IN THE*, pages 61–80. Springer, 2009.

[85] Henry Prakken and Giovanni Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of applied non-classical logics*, 7(1-2):25–75, 1997.

[86] Henry Prakken and Gerard Vreeswijk. Logics for defeasible argumentation.

[87] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[88] Iyad Rahwan, Bita Banihashemi, Chris Reed, Douglas Walton, and Sherief Abdallah. Representing and classifying arguments on the semantic web. *The Knowledge Engineering Review*, 26(4):487–511, 2011.

[89] Iyad Rahwan and Chris Reed. The argument interchange format. In *Argumentation in artificial intelligence*, pages 383–402. Springer, 2009.

[90] Iyad Rahwan and PV Sakeer. Towards representing and querying arguments on the semantic web. *Frontiers In Artificial Intelligence And Applications*, 144:3, 2006.

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

[92] Iyad Rahwan, Fouad Zablith, and Chris Reed. Laying the foundations for a world wide argument web. *Artif. Intell.*, 171(10-15):897–921, July 2007.

[93] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 2000.

[94] Chris Reed and Glenn Rowe. Araucaria: Software for argument analysis, diagramming and representation. *International Journal of AI Tools*, 14:961–980, 2004.

[95] Chris Reed and Glenn Rowe. Araucaria: Software for argument analysis, diagramming and representation. *International Journal on Artificial Intelligence Tools*, 13(04):961–979, 2004.

[96] Chris Reed and Douglas Walton. Argumentation schemes in dialogue. 2007.

[97] Chris Reed, Simon Wells, Katarzyna Budzynska, and Joseph Devereux. Building arguments with argumentation: the role of illocutionary force in computational models of argument. In *COMMA*, pages 415–426, 2010.

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

[99] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132, 1980.

[100] Ruty Rinott, Lena Dankin, Carlos Alzate Perez, Mitesh M. Khapra, Ehud Aharoni, and Noam Slonim. Show me your evidence - an automatic method for context dependent evidence detection. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 440–450, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[101] Nicolás D Rotstein, Alejandro Javier García, and Guillermo Ricardo Simari. Reasoning from desires to intentions: A dialectical framework. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 136. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[102] Sonia Vivian Rueda, Alejandro Javier García, and Guillermo Ricardo Simari. Argument-based negotiation among bdi agents. *Journal of Computer Science & Technology*, 2, 2002.

[103] Universite Paul Sabatier. Practical first-order argumentation.

[104] David C Schneider, Christian Voigt, and Gregor Betz. Argunet-a software tool for collaborative argumentation analysis and research. 2007.

[105] Jodi Schneider, Tudor Groza, and Alexandre Passant. A review of argumentation for the social semantic web. *Semant. web*, 4(2):159–218, April 2013.

[106] Christian Stab, Johannes Daxenberger, Chris Stahlhut, Tristan Miller, Benjamin Schiller, Christopher Tauchmann, Steffen Eger, and Iryna Gurevych. ArgumenText: Searching for arguments in heterogeneous sources. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 21–25, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[107] Reid Swanson, Brian Ecker, and Marilyn Walker. Argument mining: Extracting arguments from online dialogue. In *Proceedings of the 16th annual meeting of the special interest group on discourse and dialogue*, pages 217–226, 2015.

[108] Yuqing Tang, Kai Cai, Elizabeth Sklar, Peter McBurney, and Simon Parsons. A system of argumentation for reasoning about trust. In *Proceedings of the 8th European Workshop on Multi-Agent Systems, Paris, France*, 2010.

[109] Alfred Tarski and J. H. Woodger. Logic, semantics, metamathematics; papers from 1923 to 1938. *Journal of Philosophy*, 55(8):351–352, 1958.

[110] Matthias Thimm. Strategic argumentation in multi-agent systems. *KI-Künstliche Intelligenz*, 28(3):159–168, 2014.

[111] Stephen E. Toulmin. *The Uses of Argument*. Cambridge University Press, July 2003.

[112] Cassia Trojahn, Paulo Quaresma, and Renata Vieira. Conjunctive queries for ontology based agent communication in mas. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, pages 829–836, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[113] Thomas L van der Weide, Frank Dignum, J-J Ch Meyer, Henry Prakken, and GAW Vreeswijk. Multi-criteria argument selection in persuasion dialogues. In *International Workshop on Argumentation in Multi-Agent Systems*, pages 136–153. Springer, 2011.

[114] Tim Van Gelder. The rationale for rationale. *Law, probability and risk*, 6(1-4):23–42, 2007.

[115] Bart Verheij. Deflog: on the logical interpretation of prima facie justified assumptions. *Journal of Logic and Computation*, 13(3):319–346, 2003.

[116] Gerard A.W. Vreeswijk. Abstract argumentation systems. *Artificial Intelligence*, 90(1-2):225 – 279, 1997.

[117] Henning Wachsmuth, Martin Potthast, Khalid Al-Khatib, Yamen Ajjour, Jana Puschmann, Jiani Qu, Jonas Dorsch, Viorel Morari, Janek Bevendorff, and Benno Stein. Building an Argument Search Engine for the Web. In *4th Workshop on Argument Mining (ArgMining 2017) at EMNLP*, pages 49–59. Association for Computational Linguistics, September 2017.

[118] Douglas Walton. *Relevance in argumentation*. Routledge, 2003.

[119] Douglas Walton. Argumentation theory: A very short introduction. In *Argumentation in artificial intelligence*, pages 1–22. Springer, 2009.

[120] Douglas Walton and Erik CW Krabbe. *Commitment in dialogue: Basic concepts of interpersonal reasoning*. SUNY press, 1995.

[121] Douglas Walton, Christopher Reed, and Fabrizio Macagno. *Argumentation schemes*. Cambridge University Press, 2008.

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

[123] Guohui Xiao, Dag Hovland, Dimitris Bilidas, Martin Rezk, Martin Giese, and Diego Calvanese. Efficient ontology-based data integration with canonical iris. In *European Semantic Web Conference*, pages 697–713. Springer, 2018.

[124] Dimitra Zografistou, Giorgos Flouris, Theodore Patkos, and Dimitris Plexousakis. Implementing the argql query language. In *Computational Models of Argument - Proceedings of COMMA 2018, Warsaw, Poland, 12-14 September 2018*, pages 241–248, 2018.

[125] Dimitra Zografistou, Giorgos Flouris, Theodore Patkos, and Dimitris Plexousakis. A language for graphs of interlinked arguments. *ERCIM News*, 2019(118), 2019.

[126] 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.

[127] Zotero-Group. Debategraph: The global debate map, 2011.

.1 Code

.1.1 RDF data

At this point we present the RDF code for each of the AIF+ graphs in the table 4.1. We define the namespaces *aif: "http://www.arg.dundee.ac.uk/aif#"* and the common one *rdf:"http://www.w3.org/1999/02/22-rdf-syntax-ns#"*

Proposition $p \in \mathcal{P}$	$\lVert p \rVert$ = <NamedIndividual rdf:about=$u(\hat{p})$> <rdf:type rdf:resource=aif:I-node/> <aif:claimText> p </aif:claimText> </NamedIndividual>
Argument $a = \langle \{p_1, \ldots, p_n\}, c \rangle$ with $p_1, \ldots, p_n, c \in \mathcal{P}$ and $c \in Cn(\{p_1, \ldots, p_n\})$	$\lVert a \rVert$ = $\lVert p_1 \rVert \cup \cdots \cup \lVert p_n \rVert \cup \lVert c \rVert \cup$ <NamedIndividual rdf:about= $u(\hat{a})$> <rdf:type rdf:resource="aif:RA-node"/> <aif:premise rdf:resource=$u(\hat{p_1})$/> .... <aif:premise rdf:resource=$u(\hat{p_n})$/> <aif:conclusion rdf:resource=$u(\hat{c})$)/> </NamedIndividual>
Conflict $c = (p_1, p_2)$ with $p_1, p_2 \in \mathcal{P}$ and $p_1 \nvdash p_2$	$\lVert c \rVert$ = <NamedIndividual rdf:about=$u(\hat{cf})$ <rdf:type rdf:resource=aif:CA-node/> <aif:premise rdf:resource=$u(\hat{p_1})$ /> <aif:conclusion rdf:resource=$u(\hat{p_1})$ /> <aif:premise rdf:resource=$u(\hat{p_2})$ /> <aif:conclusion rdf:resource=$u(\hat{p_2})$ /> </NamedIndividual>

Table 1: Data mapping RDF

Equivalence $eq = (p_1, p_2)$ with $p_1, p_2 \in \mathcal{P}$ and $p_1 \equiv p_2$	$\lVert eq \rVert = \lVert p_1 \rVert \cup \lVert p_2 \rVert \cup$
	`<NamedIndividual rdf:about=`$u(ya_2)$
	    `<rdf:type rdf:resource=aif:YA-node/>`
	    `<aif:illocContent rdf:resource=`$u(\hat{p}_1)$ `/>`
	    `<aif:locution rdf:resource=`$u(loc_1)$ `/>`
	`</NamedIndividual>`
	`<NamedIndividual rdf:about=`$u(loc_1)$
	    `<rdf:type rdf:resource=aif:L-node/>`
	    `<aif:locution rdf:resource=`$u(ya_2)$ `/>`
	`</NamedIndividual>`
	`<NamedIndividual rdf:about=`$u(ya_3)$
	    `<rdf:type rdf:resource=aif:YA-node/>`
	    `<aif:illocutionaryContent rdf:resource=`$u(\hat{p}_2)$
	`/>`
	    `<aif:locution rdf:resource=`$u(loc_2)$ `/>`
	`</NamedIndividual>`
	`<NamedIndividual rdf:about=`$u(loc_2)$
	    `<rdf:type rdf:resource=aif:L-node/>`
	    `<aif:locution rdf:resource=`$u(ya_3)$ `/>`
	`</NamedIndividual>`
	`<NamedIndividual rdf:about=`$u(t)$
	     `<rdf:type rdf:resource=aif:T-node/>`
	     `<aif:startLocution rdf:resource=`$u(loc_1)$ `/>`
	     `<aif:endLocution rdf:resource=`$u(loc_1)$ `/>`
	     `<aif:startLocution rdf:resource=`$u(loc_2)$ `/>`
	    `<aif:endLocution rdf:resource=`$u(loc_2)$ `/>`
	`</NamedIndividual>`
	`<NamedIndividual rdf:about=`$u(ya_1)$
	     `<rdf:type rdf:resource=aif:YA-node/>`
	    `<aif:anchor rdf:resource=`$u(t)$ `/>`
	    `<aif:illocutionaryContent rdf:resource=`$u(ma)$
	`/>`
	`</NamedIndividual>`
	 `<NamedIndividual rdf:about=`$u(\hat{eq})$
	    `<rdf:type rdf:resource=aif:MA-node/>`
	`</NamedIndividual>`

Table 2: Data mapping RDF

.2 Proofs

Lemma 1. *Given a conflict pattern conflp, it holds that* $\{\|\hat{c}\| \mid c \in cf(A)\} = \mathcal{E}_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle)$

Proof. In order to prove that $\{\|\hat{c}\| \mid c \in cf(A)\} = \mathcal{E}_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle)$, we need to show that $\{\|\hat{c}\| \mid c \in cf(A)\} \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle)$ (1) and $\mathcal{E}_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle) \subseteq \{\|\hat{c}\| \mid c \in cf(A)\}$ (2).

(1) Take a conflict $c = (p_1, p_2) \in cf(A)$. From table 4.1, we have that:

$\|\hat{c}\| = \{$ *$(u(\hat{c})$ type CA-node), $(u(\hat{c})$ premise $u(\hat{p}_1))$, $(u(\hat{c})$ conclusion $u(\hat{p}_1))$, $(u(\hat{c})$ premise $u(\hat{p}_2))$, $(u(\hat{c})$ conclusion $u(\hat{p}_2))$* $\}$

Obviously, recalling the definition of $\|A\|$ we have that $\|\hat{c}\| \subseteq \|A\|$. Moreover from table 4.2 (rule r_5), we have that:

$\langle\!\langle \textbf{conflp} \rangle\!\rangle$ = *$(w_{ca}$ type CA-node) $\wedge$ $(w_{ca}$ premise $w_{i1})$ $\wedge$ $(w_{ca}$ conclusion $w_{i1})$ $(w_{ca}$ premise $w_{i2})$ $\wedge$ $(w_{ca}$ conclusion $w_{i2})$*

Given the above, for $\sigma = \{(w_{i1}, u(\hat{p}_1)), (w_{i2}, u(\hat{p}_2)), (w_{ca}, u(\hat{c}))\}$, we get that $\sigma \in \varepsilon_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle)$ and $\sigma(\langle\!\langle conflp \rangle\!\rangle) = \|\hat{c}\|$, and therefore by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|\hat{c}\| \in \mathcal{E}_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle conflp \rangle\!\rangle)$. By the definition of $\langle\!\langle conflp \rangle\!\rangle$ given above, it is obvious that g is of the form:

$g = \{$ *$(u(ca)$ type CA-node), $(u(ca)$ premise $u(i_1))$, $(u(ca)$ conclusion $u(i_1))$, $(u(ca)$ premise $u(i_2))$, $(u(ca)$ conclusion $u(i_2))$*$\}$

where u_{ca} is a uri of a CA instance, and $u(i1), u(i_2)$ are the uris of two I-node instances. Moreover $g \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g if there is a conflict $c \in cf(A)$, such that $c = (p_1, p_2)$ where $u(\hat{p}_1) = u_{i1}$ and $u(\hat{p}_2) = u_{i2}$. We note that $\|\hat{c}\| = g$, therefore $g \in \{\|\hat{c}\| \mid c \in cf(A)\}$ and thus (2) holds. □

We recall the notation $ca(i_1, i_2)$ in section 4.1.1, with which we refer to the *conflict application* between the two I-nodes i_1, i_2.

Lemma 2. *Given an equivalence pattern eqp, it holds that* $\{\|\hat{e}\| \mid e \in eq(A)\} = \mathcal{E}_{\|A\|}(\langle\!\langle eqp \rangle\!\rangle)$

Proof. In order to prove that $\{\|\hat{e}\| \mid e \in eq(A)\} = \mathcal{E}_{\|A\|}(\langle\!\langle eqp \rangle\!\rangle)$, we need to show that $\{\|\hat{e}\| \mid e \in eq(A)\} \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle eqp \rangle\!\rangle)$ (1) and $\mathcal{E}_{\|A\|}(\langle\!\langle eqp \rangle\!\rangle) \subseteq \{\|\hat{e}\| \mid e \in eq(A)\}$ (2).

(1) Take an equivalence $e = (p_1, p_2) \in eq(A)$. From table 4.1, we have that:

$\|\hat{e}\|$ = *$(u(ya_1)$ illocutionaryContent $u(\hat{p}_1))$, $(u(ya_2)$ illocutionaryContent $u(\hat{p}_2))$, $(u(ya_1)$ type YA-node), $(u(ya_2)$ type YA-node), $(u(ya_1)$ locution $u(loc_1))$, $(u(ya_2)$ locution $u(loc_2))$, $(u(loc_1)$ type L-node), $(u(loc_2)$ type L-node), $(u(ta)$ type TA-node), $(u(ta)$ startLocution $u(loc_1)$), $(u(ta)$ endLocution $u(loc_1)$), $(u(ta)$ start-Locution $u(loc_2)$), $(u(ta)$ endLocution $u(loc_2)$), $(u(ya_3)$ type YA-node), $(u(ta)$ anchor $u(ya_3))$, $(u(ya_3)$ illocutionaryContent $u(\hat{e}))$, $(u(\hat{e})$ type MA-node)*

Obviously, recalling the definition of $\|A\|$ we have that $\|\hat{e}\| \subseteq \|A\|$. Moreover from table 4.2 (rule r_6), we have that:

*$\langle\langle$**eqp**$\rangle\rangle$ = (w_{ya1} illocutionaryContent w_{i1}) ∧ (w_{ya2} illocutionaryContent w_{i2}) ∧ (w_{ya1} type YA-node) ∧ (w_{ya2} type YA-node) ∧ (w_{ya1} locution w_{loc1}) ∧ (w_{ya2} locution w_{loc2}) ∧ (w_{loc1} type L-node) ∧ (w_{loc2} type L-node) ∧ (w_{ta} type TA-node) ∧ (w_{ta} startLocution w_{loc1}) ∧ (w_{ta} endLocution w_{loc1}) ∧ (w_{ta} startLocution w_{loc2}) ∧ (w_{ta} endLocution w_{loc2}) ∧ (w_{ya2} illocutionaryContent w_{i2}) ∧ (vya3 type YA-node) ∧ (w_{ta} anchor w_{ya3}) ∧ (w_{ya3} illocutionaryContent w_{ma}) ∧ (w_{ma} type MA-node)*

Given the above, for $\sigma = \{(w_{i1}, u(\hat{p_1})), (w_{i2}, u(\hat{p_2})), (w_{ya1}, u(ya_1)), (w_{ya2}, u(ya_2)), (w_{ya3}, u(ya_3)), (w_{loc_1}, u(loc_1)), (w_{loc_2}, u(loc_2)), (w_{ta}, u(ta)), (w_{ma}, u(ma))\}$, we get that $\sigma \in \varepsilon_{\|A\|}(\langle\langle eqp \rangle\rangle)$ and $\sigma(\langle\langle eqp \rangle\rangle) = \|\hat{e}\|$, and therefore by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|\hat{e}\| \in \mathcal{E}_{\|A\|}(\langle\langle eqp \rangle\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle eqp \rangle\rangle)$. By the definition of $\langle\langle eqp \rangle\rangle$ given above, it is obvious that g is of the form:

g = { ($u(ya_1)$ illocutionaryContent $u(i_1)$)), ($u(ya_2)$ illocutionaryContent $u(i_2)$)), ($u(ya_1)$ type YA-node), ($u(ya_2)$ type YA-node), ($u(ya_1)$ locution $u(loc_1)$), ($u(ya_2)$ locution $u(loc_2)$), ($u(loc_1)$ type L-node), ($u(loc_2)$ type L-node), ($u(ta)$ type TA-node), ($u(ta)$ startLocution $u(loc_1)$), ($u(ta)$ endLocution $u(loc_1)$), ($u(ta)$ start-Locution $u(loc_2)$), ($u(ta)$ endLocution $u(loc_2)$), ($u(ya_3)$ type YA-node), ($u(ta)$ anchor $u(ya_3)$), ($u(ya_3)$ illocutionaryContent $u(ma)$), ($u(ma)$ type MA-node) }

where $u(i_1)$, $u(i_2)$, u_{ma}, $u(ya_1)$, $u(ya_2)$, $u(ya_3)$, $u(loc_1)$, $u(loc_2)$,$u(ta)$ are uris for instances of I-,MA- YA-,L-TA-nodes respectively. Moreover $g \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g if there is an equivalence $e \in eq(A)$, such that $e = (p_1, p_2)$ where $u(\hat{p_1}) = u(i_1)$, $u(\hat{p_2}) = u(i_2)$ and $u(\hat{e}) = u(ma)$. We note that $\|\hat{e}\| = g$, therefore $g \in \{\|\hat{e}\| \mid e \in eq(A)\}$ and thus (2) holds.

□

We recall the notation $df(i_1, val_1, i_2, val_2)$ in section 4.1.1, with which we refer to the *default rephrase*, between i_1, i_2, with *claimText* values val_1, val_2, respectively.

Theorem 10. *Given A an argument base, M the set with the available patterns of ArgQL, $\|\cdot\|$ and $\langle\langle\cdot\rangle\rangle$ the mappings for the data model and ArgQL patterns respectively, $I_A(\cdot)$ the interpretation function of ArgQL patterns in the argument base A and $\mathcal{E}_G(\cdot)$ the evaluation function of SPARQL graph patterns in an RDF graph G, we have that for any ArgQL pattern $m \in M$, it holds that*

$$\|I_A(m)\| = \mathcal{E}_{\|A\|}(\langle\langle m \rangle\rangle)$$

Proof. The proof works recursively for the different pattern types by decomposing the most general patterns to the ones that construct them. To this end, we follow a bottom-up approach, starting from the base type, which is the proposition pattern and building the proof upwards until we reach the most general case, which is the dialogue pattern.

In order to prove the equivalence $\|I_A(m)\| = \mathcal{E}_{\|A\|}(\langle\!\langle m\rangle\!\rangle)$ for a particular pattern type $m \in M$, we must both show that $\|I_A(m)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle m\rangle\!\rangle)$ (1) and $\mathcal{E}_{\|A\|}(\langle\!\langle m\rangle\!\rangle) \subseteq \|I_A(m)\|$ (2).

Proposition pattern

Let prp be a proposition pattern of proposition $p \in \mathcal{P}$. By the semantics of section 3.3.3, we have that

$I_A(prp) = \{p' \in props(A) \mid p' \equiv p\}$

for which it holds that $\|I_A(prp)\| = \{\|p\| \mid p \in I_A(prp)\}$

Moreover from table 4.2 (rule r_7), we have that:

$\langle\!\langle \boldsymbol{prp}\rangle\!\rangle$ = *(w_i type I-node)* $\wedge$ *(w_i claimText p)* and

$\langle\!\langle \boldsymbol{prp}\rangle\!\rangle.\langle\!\langle \boldsymbol{eqp}\rangle\!\rangle = \langle\!\langle prp\rangle\!\rangle \wedge \langle\!\langle eqp\rangle\!\rangle_{w_{i1}=\langle\!\langle prp\rangle\!\rangle.w_i}$

For the given pattern type we broke its translation in two different cases as shown before. The precise translation of a proposition pattern, would be the following:

$\langle\!\langle prp\rangle\!\rangle$ *UNION* $\langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle$

as they were defined above.

From the SPARQL semantics, we have that

$\mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle \text{ UNION } \langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle) = \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle)$

thus, it suffices to show that:

$\|I_A(prp)\| = \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle)$

which means that both of the $\|I_A(prp)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle)$ (1) and
$\mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle) \subseteq \|I_A(prp)\|$ (2) must be satisfied.

(1) For the left to right direction, we assume a proposition $p \in I_A(prp)$. We discern the cases that $prp = p$, or $prp = p'$ for some $p' \equiv p$. In particular:

a) If $prp = p$ then from table 4.1, we have that:

$\|\hat{p}\|$= { *($u(\hat{p})$ type I-node), ($u(\hat{p})$ claimText p)*}

Recalling the definition of $\|A\|$ we have that $\|\hat{p}\| \subseteq \|A\|$.

Given the above, for $\sigma = \{(w_i, u(\hat{p}))\}$, we get that $\sigma \in \varepsilon_{\|A\|}(\langle\!\langle prp\rangle\!\rangle)$ and $\sigma(\langle\!\langle prp\rangle\!\rangle) = \|\hat{p}\|$, and therefore by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|\hat{p}\| \in \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle)$. Thus (1) holds.

b) If $prp = p'$ for some $p' \equiv p$, it means that there exist some $e = (p, p') \in eq(A)$. From table 4.1, we have that

$\|\hat{p'}\|$= { *($u(\hat{p'})$ type I-node), ($u(\hat{p'})$ claimText p')*}

Recalling the definition of $\|A\|$, we have that $\|\hat{p'}\| \cup \|\hat{e}\| \subseteq \|A\|$.

Given the above, for $\sigma = \{(w_i, u(\hat{p'})), (\langle\!\langle eqp \rangle\!\rangle.w_{i2}, u(\hat{p})), (\langle\!\langle eqp \rangle\!\rangle.w_{ma}, u(\hat{e}))\}$, we get that $\sigma \in \varepsilon_{\|A\|}(\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle)$ and $\sigma(\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle) = \|\hat{p}\| \cup \|\hat{e}\|$ and therefore by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|\hat{p'}\| \cup \|\hat{e}\| \in \mathcal{E}_{\|A\|}(\langle\!\langle prp \rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle)$. Thus (1) holds also for this case.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle prp \rangle\!\rangle) \cup \mathcal{E}_{\|A\|}(\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle)$. We discern two cases:

a) If $g \in \mathcal{E}_{\|A\|}(\langle\!\langle prp \rangle\!\rangle)$, then g will be of the form:

$g = \{$ *(*$u(i)$ *type I-node), (*$u(i)$ *claimText* p'*)* $\}$

where $u(i)$ the uri of an I-node instance i. Moreover $g \subseteq \|A\|$. The $\|A\|$, created by the translation of A can only contain g if there is a proposition $p \in \mathcal{P}(A)$, the value of which is *txtVal*, where $u(\hat{p}) = u(i)$. We note that $\|\hat{p}\| = g$, therefore $g \in \|I_A(prp)\|$. Thus (2) holds.

b) If $g \in \mathcal{E}_{\|A\|}(\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle)$, then g will be of the form:

$g = \{$ *(*$u(i)$ *type I-node), (*$u(i)$ *claimText p),* $df(i, p, i', p')$ $\}$

where $u(i), u(i')$ are uris of two I instances i and i' and $df(i, p, i', p')$ is a default rephrase between them. Moreover $g \subseteq \|A\|$.

The $\|A\|$, created by the translation of A can only contain g if there is a proposition $p \in props(A)$, where $u(\hat{p}) = u(i)$ and also if there is an equivalence $e = (p, p')$, such that $\|\hat{e}\| = df(i, p, i', p')$, $u(\hat{p'}) = u(i')$ where $p' \in props(a)$ a second proposition. We have that both $p, p' \in I_A(prp)$. We note that $\|\hat{p}\| \cup \|\hat{e}\| = g$, therefore $g \in \|I_A(prp)\|$. Thus (2) holds.

Conclusion pattern

We discern the cases where a conclusion pattern *conclp* is a proposition pattern or it is a variable.

- If *conclp* is a constant proposition pattern *prp* of a proposition $p \in \mathcal{P}$, by the semantics of section 3.3.3, we have that:

$I_A(conclp) = \{c \in I_A(prp) \mid concl(a) = c$ for some $a \in A\}$

Moreover, from table 4.3 (rule r_{10}.1), we have that:

$\langle\!\langle$***conclp***$\rangle\!\rangle = \big(\ \langle\!\langle prp \rangle\!\rangle \wedge$ *(*w_{ra} *conclusion* $\langle\!\langle prp \rangle\!\rangle.w_i$*)* *UNION* $\langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{w_{i2}=w_i} \wedge ($$w_{ra}$ *conclusion* w_i*)* $\big)$ $\wedge$ *(*w_{ra} *type RA-node)*

We decompose and rewrite $\langle\!\langle conclp \rangle\!\rangle$ into smaller graph patterns as follows:

$gp_1 = \langle\!\langle prp \rangle\!\rangle \wedge$ *(*w_{ra} *conclusion* $\langle\!\langle prp \rangle\!\rangle.w_i$*)* $\wedge$ *(*w_{ra} *type RA-node)*

$gp_2 = \langle\!\langle prp \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{w_{i2}=w_i} \wedge ($$w_{ra}$ *conclusion* w_i*)*$\wedge$ *(*w_{ra} *type RA-node)*

such that $\langle\!\langle conclp \rangle\!\rangle = gp_1$ *UNION* gp_2

From the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(gp_1\ UNION\ gp_2) = \sigma \in \varepsilon_A(gp_1)\ or\ \sigma \in \varepsilon_A(gp_2)$

- $\mathcal{E}_{\|A\|}(gp_1\ UNION\ gp_2) = \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2)$

(1) For the left to right direction, take a proposition $c \in I_A(conclp)$. This means that $\exists a \in A : concl(a) = c$ and $p = c$, or $p \equiv c$. According to the table 4.1, by breaking the case of an argument into its components *prem(a)* and *concl(a)*, we have that:

$\|concl(a)\| = \{\|c\| \cup \{(u(\hat{a})\ conclusion\ u(\hat{c})), (u(\hat{a})\ type\ RA\text{-}node)\}\}$

Obviously, and given that $\|I_A(conclp)\| = \{\|c\| \mid c \in I_A(conclp)\}$ we have that $\|concl(a)\| \in \|I_A(conclp)\|$ and also, recalling the definition of $\|A\|$, we have that: $\|concl(a)\| \subseteq \|A\|$

Given the above and according to the proof of the previous case (proposition pattern), if $c = p$, then for $\sigma = \{\ (\langle\langle prp\rangle\rangle.w_i, u(\hat{c})),\ (w_{ra}, u(\hat{a}))\ \}$, we get that

$\sigma \in \varepsilon_{\|A\|}(gp_1)$ and
$\sigma(gp_1) = \{\|c\| \cup \{(u(\hat{a})\ conclusion\ u(\hat{c})), (u(\hat{a})\ type\ RA\text{-}node\ )\}\}$

and therefore by the definition of $\mathcal{E}_{\mathcal{G}}$ $\|c\| \in \mathcal{E}_{\|A\|}(gp_1)$. Thus (1) holds.

On the other hand, if $c \equiv p$, such that $e = (c, p) \in eq(A)$, then for $\sigma = \{\ (\langle\langle prp\rangle\rangle.\langle\langle eqp\rangle\rangle.w_{i2}, u(\hat{c})),\ (\langle\langle prp\rangle\rangle.w_i, u(\hat{p})),\ (\langle\langle prp\rangle\rangle.\langle\langle eqp\rangle\rangle.w_{ma}, u(\hat{e})),\ (w_{ra}, u(\hat{a}))\ \}$, we get that

$\sigma \in \varepsilon_{\|A\|}(gp_2)$ and
$\sigma(gp_2) = \{\|p\| \cup \|e\| \cup \{(u(\hat{a})\ conclusion\ u(\hat{c})),\ (u(\hat{a})\ type\ RA\text{-}node)\}$

and therefore by the definition of $\mathcal{E}_{\mathcal{G}}$:

$\{\|p\| \cup \|e\| \cup (u(\hat{a})\ conclusion\ u(\hat{c})) \cup (u(\hat{a})\ type\ RA-node)\} \in \mathcal{E}_{\|A\|}(gp_2)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle conclp\rangle\rangle)$. Because of the existence of UNION, we discern two cases:

a) If $g \in gp_1$, then g will be of the form:

$g = \{\ (u(i)\ type\ I\text{-}node),\ (u(i)\ claimText\ p),\ (u(ra)\ conclusion\ u(i),\ (u(ra)\ type\ RA\text{-}node)\}$

where $u(i)$ is a uri of an I-node instance i and $u(ra)$ is the uri of an RA-node instance. Moreover, $g \subseteq \|A\|$. $\|A\|$, created by the translation of A can only contain g if there is a proposition $p \in \mathcal{P}(A)$, where $u(\hat{p}) = u(i)$, and also, since there is the triple $(u(ra)\ conclusion\ u(i))$, it also means that $p = concl(a)$ for some $a \in A$, for which $u(\hat{a}) = u(ra)$. As a result we infer that $\{\|\hat{p}\| \cup (u(\hat{a})\ conclusion\ u(\hat{p}), (u(ra)\ type\ RA\text{-}node)\ \} = g$, therefore $g \in \|I_A(conclp)\|$. Thus (2) holds.

b) If $g \in gp_2$, then g will be of the form:

$g = \{\ (u(i)\ type\ I\text{-}node),\ (u(i)\ claimText\ p),\ df(i, p, i', c)\ (u(ra)\ conclusion\ u(i')\ ),\ (u(ra)\ type\ RA\text{-}node)$
$\}$

where $u(i), u(i')$ are uris of two I instances i and i', $u(ra)$ is the uri of an RA-node instance and $df(i,p,i',c)$ is a default rephrase between i and i'. Moreover, $g \subseteq \|A\|$. $\|A\|$, created by the translation of A can only contain g if the following are satisfied by A:

- There is a proposition $p \in props(A)$, where $u(\hat{p}) = u(i)$ and value *cval*
- There is an equivalence $e = (p,c)$, such that $\|\hat{e}\| = df(i,p,i',c)$, $u(\hat{c}) = u(i')$ where $c \in props(a)$ a second proposition with value *cval'*.
- For proposition c, it also holds that $c = concl(a)$, for some $a \in A$, for which $u(\hat{a}) = u(ra)$.

As a result we infer that $\{\|\hat{c}\| \cup \|\hat{e}\| \cup (u(\hat{a})\ conclusion\ u(\hat{c})\} = g$, therefore $g \in \|I_A(conclp)\|$. Thus (2) holds.

- If the conclusion pattern is a variable v, we have:

$I_A(conclp) = \{\mu(v) \in props(A) \mid \mu(v) = concl(a)$, for some $a \in A\}$

Moreover, from table 4.3 (rule $r_{10}.2$), we have that:

$\langle\langle \textbf{conclp} \rangle\rangle = (w_i\ type\ I\text{-}node\ ) \wedge (w_i\ claimText\ v_c) \wedge (w_{ra}\ conclusion\ w_i) \wedge (w_{ra}\ type\ RA\text{-}node)$

(1) For the left to right direction, take a proposition $c \in I_A(conclp)$. This means that $\exists a \in A : concl(a) = c$ According to table 4.1, we have that:

$\|concl(a)\| = \{\|c\| \cup \{(u(\hat{a})\ conclusion\ u(\hat{c})), (u(\hat{a})\ type\ RA\text{-}node)\}\}$

Obviously we have that $\|concl(a)\| \in \|I_A(conclp)\|$ and also, recalling the definition of $\|A\|$, we have that: $\|concl(a)\| \subseteq \|A\|$

Given the above, for $\sigma = \{\ (w_i, u(\hat{c})), (v,c), (w_{ra}, u(\hat{a}))\ \}$, we get that

$\sigma \in \varepsilon_{\|A\|}(\langle\langle conclp \rangle\rangle)$ and
$\sigma(\langle\langle conclp \rangle\rangle) = \{\|c\| \cup (u(\hat{a})\ conclusion\ u(\hat{c})) \cup (u(\hat{a})\ type\ RA\text{-}node)\}$

and therefore by the definition of $\mathcal{E}_G$: $\|concl(a)\| \in \mathcal{E}_{\|A\|}(\langle\langle conclp \rangle\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle conclp \rangle\rangle)$. By the definition of $\langle\langle conclp \rangle\rangle$ given above, it is obvious that g will have the following form:

$g = \{\ (u(i)\ type\ I\text{-}node), (u(i)\ claimText\ c), (u(ra)\ conclusion\ u(i)), (u(ra)\ type\ RA\text{-}node)\}$

where $u(i)$ the uri of an I instance i and $u(ra)$ is the uri of an RA-node instance. Moreover, $g \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g if there is a proposition $c \in \mathcal{P}(A)$, such that $u(\hat{c}) = u(i)$, and also, since there are the triples *(*$u(ra)$ *conclusion* $u(i)$*), (*$u(ra)$ *type RA-node)*, it also means that $c = concl(a)$ for some $a \in A$, for which $u(\hat{a}) = u(ra)$.

As a result we infer that $\{\|\hat{c}\| \cup (u(\hat{a})\ \textit{conclusion}\ u(\hat{c}) \cup \textit{(}u(ra)\ \textit{type RA-node)}\} = g$, therefore $g \in \|I_A(conclp)\|$. Thus (2) holds.

Premise pattern

We discern the cases where a premise pattern *premp* is a set of proposition patterns or it has the form $v[f_{pr}]$, where $v \in V$ and f_{pr} is a premise filter.

- If *premp* is a set of proposition patterns $s = \{prp_1, \dots, prp_n\}$, of propositions $p_1, \dots, p_n$, respectively, then, by the semantics of section 3.3.3, we have:

$I_A(premp) = \{\{p'_1, \dots p'_n\} \mid p'_i \in I_A(prp_i)$ and $\{p'_1, \dots p'_n\} = prem(a)$ for some $a \in A\}$

Moreover, from table 4.3 (rule r_9.1), we have that:

$$\langle\!\langle \textbf{\textit{premp}} \rangle\!\rangle = \Big[\bigwedge_{j=1}^{n} \big(\ \langle\!\langle prp_j \rangle\!\rangle \wedge (\langle\!\langle prp_j \rangle\!\rangle.w_i\ \textit{premise}\ w_{ra})\big)\ \textit{UNION}\ \big(\langle\!\langle prp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{w_{i2}=w_{ij}}\ (w_{ij}\ \textit{premise}\ w_{ra})\ \big)\Big]$$

$$\wedge\ \textit{NOT EXISTS}\ \big((\bigwedge_{j=1}^{n} w_x \neq \langle\!\langle prp_j \rangle\!\rangle.w_i)\ \ \wedge\ \ (w_x\ \textit{premise}\ w_{ra})\ \big) \wedge (w_{ra}\ \textit{type RA-node})$$

We decompose $\langle\!\langle premp \rangle\!\rangle$ into smaller graph patterns and rewrite it as follows (using the distributive property: $(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$):

$gp_j^1 = \langle\!\langle prp_j \rangle\!\rangle \wedge (\langle\!\langle prp_j \rangle\!\rangle.w_i\ \textit{premise}\ w_{ra}) \wedge (w_{ra}\ \textit{type RA-node})$

$gp_j^2 = \langle\!\langle prp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{w_{i2}=w_{ij}}\ (w_{ij}\ \textit{premise}\ w_{ra}) \wedge (w_{ra}\ \textit{type RA-node})$

$gp_j = gp_j^1\ \textit{UNION}\ gp_j^2$

$expr = \text{NOT EXISTS}\ \big((\bigwedge_{j=1}^{n} w_x \neq \langle\!\langle prp_j \rangle\!\rangle.w_i)\ \ \wedge\ \ (w_x\ \text{premise}\ w_{ra})\big)$

such that $\langle\!\langle premp \rangle\!\rangle = [\ \bigwedge_{j=1}^{n} gp_j^1\ \textit{UNION}\ gp_j^2\ ] \wedge expr$

From the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(gp_j) = \sigma \in \varepsilon_A(gp_j^1)\ \textit{or}\ \sigma \in \varepsilon_A(gp_j^2)$

- $\mathcal{E}_{\|A\|}(gp_j) = \mathcal{E}_{\|A\|}(gp_j^1) \cup \mathcal{E}_{\|A\|}(gp_j^2)$

- $\sigma \vDash_S expr =¿\ \sigma \vDash_S \textit{NOT EXISTS}\ \big((\bigwedge_{j=1}^{n} w_x \neq \langle\!\langle prp_j \rangle\!\rangle.w_i)\ \ \wedge\ \ (w_x\ \textit{premise}\ w_{ra})\big) =¿$

$\sigma \nvDash_S (w_x \neq \langle\!\langle prp_1 \rangle\!\rangle.w_i\ \textit{and} \dots \textit{and}\ w_x \neq \langle\!\langle prp_n \rangle\!\rangle.w_i\ \textit{and}\ (w_x\ \textit{premise}\ w_{ra}))$

- $\mathcal{E}_{\|A\|}(premp) = \{\{\sigma(gp_1), \dots, \sigma(gp_n)\} \mid \sigma \in (\varepsilon_A(gp_1) \bowtie \cdots \bowtie \varepsilon_A(gp_n))$ *and it is not the case that* $\sigma \vDash_S expr\ \}$

(1) For the left to right direction, take a set of propositions $\{p'_1, \dots, p'_n\} \in I_A(premp)$. This means that $\exists a \in A : prem(a) = \{p'_1, \dots, p'_n\}$. According to table 4.1 by breaking the case of an argument into its

components *prem(a)* and *concl(a)*, we infer that:

$\|prem(a)\|$ = { $\|p'_1\|$, *(*$u(\hat{p'_1})$ *premise* $u(\hat{a})$*),* ... $\|p'_n\|$, *(* $u(\hat{p}_n)$ *premise* $u(\hat{a})$*),* *(*$u(\hat{a})$ *type RA-node)*}

Obviously, assuming that $\|I_A(premp)\| = \{\{\|p'_1\|,(u(\hat{p'_1})$ *premise* $u(\hat{a})),\ldots,\|p'_n\|,(u(\hat{p'_n})$ *premise* $u(\hat{a})$*), (*$u(\hat{a})$ *type RA-node)* } $\mid \{p'_1,\ldots,p'_n\} \in I_A(premp)\}$

we infer $\|prem(a)\| \in \|I_A(premp)\|$ and also, recalling the definition of $\|A\|$, we have that $\|prem(a)\| \subseteq \|A\|$.

For each of the $p'_j \in prem(a)$, it holds that either $p'_j = p_j$, or $p'_j \equiv p_j$. In the first case, for σ = { ($\langle\!\langle prp_j\rangle\!\rangle$*.*$w_i$, $u(\hat{p}_j)$), (w_{ra},$u(\hat{a})$) }, we get that $\sigma \in \varepsilon_{\|A\|}(gp^1_j)$ while in the other case, for σ = { ($\langle\!\langle prp_j\rangle\!\rangle$*.*$w_i$, $u(\hat{p}_j)$), ($\langle\!\langle prp_j\rangle\!\rangle$*.*$\langle\!\langle eqp\rangle\!\rangle$*.*$w_{i2}$, $u(\hat{p'_j})$), (w_{ra},$u(\hat{a})$) } we get that $\sigma \in \varepsilon_{\|A\|}(gp^2_j)$. As a result, for both cases, there is some σ: $\sigma \in \varepsilon_{\|A\|}(gp_j)$. Given that $\forall p'_j \in prem(a)$, there is some σ that satisfies the respective gp_j and in which there exists the pair $(w_{ra},u(\hat{a}))$ we infer that for this σ: $\sigma \in \varepsilon_{\|A\|}(gp_1) \bowtie \cdots \bowtie \varepsilon_{\|A\|}(gp_n)$. Furthermore, since $prem(a) \in I_A(premp)$, according to the ArgQL semantics, we have that: $\forall p'_j \in prem(a)$, $p'_j \in I_A(prp_j)$ for $1 \le j \le n$ of the *premp*. Consequently, according to the proof of the proposition pattern case, we have that $\forall p'_j$: $\|p'_j\| \in \mathcal{E}_{\|A\|}(\langle\!\langle prp_j\rangle\!\rangle)$. This can be written differently as: it is not the case that $\exists\hat{x}$: $(u(\hat{x})$ *premise* $u(\hat{a}))$, for which there is no tuple ($\langle\!\langle prp_j\rangle\!\rangle$*.*$w_i$, $u(\hat{x})$) in the σ mapping. In this way we show that $\sigma \vDash_S expr$.

Given the above, we get that $\sigma \in \varepsilon_{\|A\|}(\langle\!\langle premp\rangle\!\rangle)$ and therefore by the definition of $\mathcal{E}_G$:

$\|prem(a)\| \in \mathcal{E}_{\|A\|}(\langle\!\langle premp\rangle\!\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle premp\rangle\!\rangle)$. By the definition of $\langle\!\langle premp\rangle\!\rangle$ given above, it is obvious that g will be:

$g = \{g_1,\ldots,g_n\}$, such that $g_j \in \mathcal{E}_{\|A\|}(gp_j)$ which also satisfies the *expr* part.

For each g_j, we discern two cases:

a) if $g_j \in \mathcal{E}_{\|A\|}(gp^1_j)$, then g will be of the form:

g_j *= (*$u(i_j)$ *type I-node), (*$u(i_j)$ *claimText* p_j*), (*$u(i_j)$ *premise* $u(ra)$*), (*$u(ra)$ *type RA-node)*

where $u(i_j)$ is the uri of an I-node instance, and $u(ra)$ the uri of an RA-node instance. Moreover $g_j \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g_j if there is a proposition p_j, where $u(\hat{p}_j) = u(i_j)$, for which it holds that $p_j \in prem(a)$, for some argument $a \in A$, for which $u(\hat{a}) = u(ra)$.

b) Respectively, if $g_j \in \mathcal{E}_{\|A\|}(gp^2_j)$, then g_j will be of the form:

g = { *(*$u(i_j)$ *type I-node), (*$u(i_j)$ *claimText* p_j*), df(*i_j, p_j, i'_j, p'_j*), (*$u(i'_j)$ *premise* $u(ra)$*), (*$u(ra)$ *type RA-node)*}

where $u(i_j),u(i'_j)$ are uris of two I-node instances i_j,i'_j, $u(ra)$ the uri of an RA-node instance and *df(*i_j, p_j, i'_j, p'_j*)* is a default rephrase between i_j and i'_j. Moreover $g \subseteq \|A\|$.

$\|A\|$, created by the translation of A can only contain g if the following are satisfied by A:

- There is a proposition $p_j \in props(A)$, where $u(\hat{p}_j) = u(i_j)$ and value val_j
- There is an equivalence $e = (p_j, p'_j)$, such that $\|\hat{e}\| = df(i_j, val_j, i'_j, cval'_j)$, $u(\hat{p'_j}) = u(i'_j)$ where $p'_j \in props(A)$ a second proposition with value val'_j.
- For proposition p'_j, it also holds that $p'_j \in prem(a)$, for some $a \in A$, for which $u(\hat{a}) = u(ra)$.

All the above are satisfied for all $1 \le j \le n$. Since all the different g_j join on the same $u(ra) = u(\hat{a})$, it means that all of the respective p_j or p'_j are premises of the same $a \in A$. Moreover, we also have that *expr* is satisfied by g, which means that there is no other I-node x for which *(u(x) premise u(ra))*, which is different than the I-nodes matching to the different gp_js indicated in $\langle\!\langle premp \rangle\!\rangle$. By extension, there must not exist no other $p_x \in props(A)$, such that $p_x \in prem(a)$, because if there was, the $\|prem(a)\|$ would also have created the triple *$(u(\hat{p}_x)$ premise $u(\hat{a}))$* in $\|A\|$.

Concluding, we infer that $\|prem(a)\| = g$ and therefore $g \in \|I_A(premp)\|$. Thus (2) holds.

- If premp has the form $v[f_{pr}]$, where $v \in V$ and f_{pr} is a premise filter. We will show the proof for all the different forms f_{pr} may get. Thus we have:

- If $premp = v\ [/\{sp_1, .., sp_m\}]$, with sp_j proposition pattern of propositions s_j.

  By the semantics of section 3.3.3, we have that:

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq props(A),\ \mu(v) = prem(a)$ for some $a \in A$ and $\{s_1, \ldots s_m\} \subseteq prem(a)$ with $s_i \in I_A(sp_i)$ and $1 \le i \le m$ $\}$

  Moreover, from table 4.2 (rules $r_8.2$ and $r_9.1$), we have that:

  $$\langle\!\langle \textbf{premp} \rangle\!\rangle = (w_i\ type\ I\text{-}node) \wedge (w_i\ claimText\ v_p) \wedge (w_i\ premise\ w_{ra}) \wedge (w_{ra}\ type\ RA\text{-}node) \wedge \langle\!\langle f_{pr} \rangle\!\rangle_{w_{ra}}$$

  $$\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle = \bigwedge_{j=1}^{m} \Big[\big(\langle\!\langle sp_j \rangle\!\rangle \wedge (\langle\!\langle sp_j \rangle\!\rangle.w_i\ premise\ w_{ra}) \big) \vee \big(\langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{wi2=w_{ij}} \wedge (w_{ij}\ premise\ w_{ra}) \big) \Big]$$

  We decompose $\langle\!\langle f_{pr} \rangle\!\rangle$ into smaller graph patterns and rewrite it as:

  $$\langle\!\langle f_{pr} \rangle\!\rangle = \bigwedge_{j=1}^{n} gp_j$$

  where $gp_j = (\ gp_j^1\ UNION\ gp_j^2\ )$

  $$gp_j^1 = \langle\!\langle sp_j \rangle\!\rangle \wedge (\langle\!\langle sp_j \rangle\!\rangle.w_i\ premise\ w_{ra}) \big)$$

  $$gp_j^2 = \langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{wi2=w_{ij}} \wedge (w_{ij}\ premise\ w_{ra})$$

Overall we have that

$\langle\!\langle \boldsymbol{premp} \rangle\!\rangle = gp \wedge \langle\!\langle f_{pr} \rangle\!\rangle$

where gp = *(w_i type I-node) ∧ (w_i claimText v_p) ∧ (w_i premise w_{ra}) ∧ (w_{ra} type RA-node)*

From the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(gp_j) = \sigma \in \varepsilon_A(gp_j^1)$ *or* $\sigma \in \varepsilon_A(gp_j^2)$

- $\mathcal{E}_{\|A\|}(gp_j) = \mathcal{E}_{\|A\|}(gp_j^1) \cup \mathcal{E}_{\|A\|}(gp_j^2)$

- $\sigma(\langle\!\langle f_{pr} \rangle\!\rangle) = \sigma(gp_1) \cup \cdots \cup \sigma(gp_m)$

- $\mathcal{E}_{\|A\|}(premp) = \{\sigma(gp) \cup \sigma(\langle\!\langle f_{pr} \rangle\!\rangle) \mid \sigma \in (\varepsilon_A(gp) \bowtie \varepsilon_A(gp_1) \bowtie \cdots \bowtie \varepsilon_A(gp_n))\}$

a) For the left to right direction, take a set of propositions $\{p_1, \ldots, p_n\} \in I_A(premp)$. This means that $\exists a \in A : prem(a) = \{p_1, \ldots, p_n\}$. In addition, for $s'_1 \in I_A(sp_1)$, …, $s'_m \in I_A(sp_m)$, it also holds $\{s'_1, \ldots, s'_m\} \subseteq prem(a)$, and without loss of generality, we have that $prem(a) = \{p_1, \ldots, s'_1, .., s'_m, .., p_n\}$.
From the table 4.1, we get that

$\|prem(a)\| = \{\ \|p_1\|, (u(\hat{p_1})$ *premise* $u(\hat{a}))$, .. , $\|s'_1\|, (u(\hat{s'_1})$ *premise* $u(\hat{a}))$, .. , $\|s'_m\|, (u(\hat{s'_m})$ *premise* $u(\hat{a}))$, …, $\|p_n\|$, $(\ u(\hat{p_n})$ *premise* $u(\hat{a}))$, $(u(\hat{a})$ *type RA-node*$)\ \}$

Obviously, we have that $\|prem(a)\| \in \|I_A(premp)\|$, and also, recalling the definition of $\|A\|$, we have that $\|prem(a)\| \subseteq \|A\|$.

Let n different σ mappings of *premp*, such that:

$\sigma_1 = \{(w_i,\ u(\hat{p_1}),\ (v,\ p_1),\ (w_{ra},\ u(\hat{a})),\ map_{\sigma_1}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )\}$

…

$\sigma_n = \{(w_i,\ u(\hat{p_n}),\ (v,\ p_n),\ (w_{ra},\ u(\hat{a})),\ map_{\sigma_n}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )\}$

$map_{\sigma_i}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ ) = \{map_{s1}, \ldots, map_{sm}, (w_{ra}, u(\hat{a}))\}$

where for $1 \leq j \leq m$, map_{sj} is the respective mapping for gp_j, depending on whether $\sigma_i \in \varepsilon_{\|KB\|}(gp_j^1)$ or $\sigma_i \in \varepsilon_{\|KB\|}(gp_j^2)$ (similar to the previous case where the premise pattern was a set of proposition patterns). In particular it holds that either $map_{sj} = (\langle\!\langle sp_j \rangle\!\rangle.w_i,\ u(\hat{s_j}))$ or $map_{sj} = \{\ (\langle\!\langle sp_j \rangle\!\rangle.w_i,\ u(\hat{s'_j})),\ (\langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle.w_{i2},\ u(\hat{s_j}))\ \}$

Given the above, we get that $\sigma_1, \ldots, \sigma_n \in \varepsilon_{\|A\|}(premp)$ and
$\|prem(a)\| \subseteq \sigma_1(\langle\!\langle premp \rangle\!\rangle) \cup \cdots \cup \sigma_n(\langle\!\langle premp \rangle\!\rangle)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$:

$\|prem(a)\| \subseteq \mathcal{E}_{\|A\|}(\langle\langle premp \rangle\rangle)$. Thus (1) holds.

2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle premp \rangle\rangle)$. By the definition of $\langle\langle premp \rangle\rangle$ g will be of the form

$g = \{(u(i)\ type\ I\text{-}node\ ),\ (u(i)\ claimText\ p),\ (u(i)\ premise\ u(ra)),\ g_{f_{pr}}\}$

where $u(i)$ is the uri of an I-node instance, $u(ra)$ the uri of an RA-node instance, and

$g_{f_{pr}} = g_1, \dots g_m$

For each g_j, we discern two cases:

a) if $g_j \in \mathcal{E}_{\|A\|}(gp^1_j)$, then g will be of the form:

g = { $(u(i_1)$ *type I-node*$)$, $(u(i_j)$ *claimText* $s_j)$, $(u(i_j)$ *premise* $u(ra))$,}

where $u(i_j)$ is the uri of an I-node instance, and $u(ra)$ the uri of an RA-node instance. Moreover $g_j \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g if there is a proposition s_j, where $u(\hat{s_j}) = u(i_j)$, for which it holds that $s_j \in prem(a)$, for some argument $a \in A$, for which $u(\hat{a}) = u(ra)$.

b) Respectively, if $g_j \in \mathcal{E}_{\|A\|}(gp^2_j)$, then g_j will be of the form:

g = { $(u(i_j)$ *type I-node*$)$, $(u(i_j)$ *claimText* $s_j)$, *df*$(i_j,\ s_j,\ i'_j,\ s'_j)$, $(u(i'_j)$ *premise* $u(ra))$}

where $u(i_j), u(i'_j)$ are uris of two I-node instances i_j, i'_j, $u(ra)$ the uri of an RA-node instance and *df*$(i_j,\ s_j,\ i'_j,\ s'_j)$ is a default rephrase between i_j and i'_j. Moreover $g \subseteq \|A\|$.

$\|A\|$, created by the translation of A can only contain g if the following are satisfied by A:

- There is a proposition $s_j \in props(A)$, where $u(\hat{s_j}) = u(i_j)$ and value val_j
- There is an equivalence $e = (s_j, s'_j)$, such that $\|\hat{e}\| = df(i_j, s_j, i'_j, s'_j)$, $u(\hat{s'_j}) = u(i'_j)$ where $s'_j \in props(A)$ a second proposition with value val'_j.
- For proposition s'_j, it also holds that $s'_j \in prem(a)$, for some $a \in A$, for which $u(\hat{a}) = u(ra)$.

All the above are satisfied for all $1 \leq j \leq m$. Since all the different g_j join on the same $u(ra) = u(\hat{a})$, it means that all of the respective s_j or s'_j are included in the premises of the same $a \in A$, otherwise, $\{s_1, \dots, s_m\} \subseteq prem(a)$ for s_j or s'_j.

Concluding, we infer that $prem(a) \in I_A(premp)$ and since $g \subseteq \|prem(a)\|$ we get that $g \subseteq \|I_A(premp)\|$. Thus (2) holds.

- If $f_{pr} = f_{incl} = [/v_2]$, where $v_2 \in V$ the main variable appearing in a different premise pattern $premp_2$ of an argument pattern ap_2.

  By the semantics of section 3.3.3, we have that:

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq P(A),\ \mu(v) = prem(a)$ for some $a \in A$, $\mu(v') = prem(a')$ for some $a' \in I_A(ap')$ and $\mu(v') \sqsubseteq \mu(v)\}$

  Moreover, from table 4.2 (rules $r_8.2$ and $r_9.2$), we have that:

  $\langle\!\langle \boldsymbol{premp} \rangle\!\rangle$ = *(w_i type I-node) ∧ (w_i claimText v_p) ∧ (w_i premise w_{ra}) ∧ (w_{ra} type RA-node) ∧* $\langle\!\langle f_{pr} \rangle\!\rangle_{w_{ra}}$

  $\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle$ = *NOT EXISTS* { (w_i *premise* $\langle\!\langle premp_2 \rangle\!\rangle.w_{ra}$) ∧

  NOT EXISTS { (w_i *premise* w_{ra}) ∨ ($\langle\!\langle eqp \rangle\!\rangle_{w_{i1}=w_i}$ ∧ ($\langle\!\langle eqp \rangle\!\rangle.w_{i2}$ *premise* w_{ra})) } }

  For gp = *(w_i type I-node) ∧ (w_i claimText v_p) ∧ (w_i premise w_{ra}) ∧ (w_{ra} type RA-node)*

  we have that

  $\langle\!\langle \boldsymbol{premp} \rangle\!\rangle = gp \wedge \langle\!\langle f_{pr} \rangle\!\rangle$

  - $\mathcal{E}_{\|A\|}(premp) = \{\sigma(gp) \mid \sigma \vDash_S \langle\!\langle f_{pr} \rangle\!\rangle\ \}$

  a) For the left to right direction, take a set of propositions $\{p_1,\ldots,p_n\} \in I_A(premp)$. This means that $\exists a \in A : prem(a) = \{p_1,\ldots,p_n\}$. In addition it also means that ap' has been evaluated and $\exists a' \in I_A(ap')$, s.t. $\mu(v') = prem(a')$ and $prem(a') \sqsubseteq prem(a)$ (here we have $\sqsubseteq$ instead of $\subseteq$). In other words, $\forall x \in prem(a')$, it holds that $x \in prem(a)$ or $\exists x' \in prem(a')$, s.t. $x \equiv x'$.

  From the table 4.1 we get that

  $\|prem(a)\| = \{\ \|p_1\|, (u(\hat{p}_1)$ *premise* $u(\hat{a}))$, …, $\|p_n\|$, $(u(\hat{p}_n)$ *premise* $u(\hat{a}))$, $(u(\hat{a})$ *type RA-node)*$\}$

  Obviously, we have that $\|prem(a)\| \in \|I_A(premp)\|$, and recalling the definition of $\|A\|$, we also have that $\|prem(a)\| \subseteq \|A\|$.

  Let n different σ mappings of $premp$, such that:

  $\sigma_1 = \{(w_i,\ u(\hat{p}_1)),\ (v,\ pval_1),\ (w_{ra},\ u(\hat{a}))\}$

  …

  $\sigma_n = \{(w_i,\ u(\hat{p}_n)),\ (v,\ pval_n),\ (w_{ra},\ u(\hat{a}))\}$

  where $1 \le i \le n$, $pval_i$ is the text value of proposition p_i and

The condition that holds in *prem(a)*, namely that $\forall x \in prem(a')$, it holds that $x \in prem(a)$ or $\exists x' \in prem(a')$, s.t. $x \equiv x'$, is preserved in the $\|prem(a)\|$, in the following way: $\forall \|x\|$ s.t. $(u(\hat{x})$ *premise* $u(\hat{a'}))$, it also holds that $(u(\hat{x})$ *premise* $u(\hat{a}))$ or there is a $df(\hat{x},x,\hat{x'},x')$ and for $\hat{x'}$, it also holds that $(u(\hat{x'})$ *premise* $u(\hat{a}))$. This is exactly what the filter in $\langle\!\langle f_{pr}\rangle\!\rangle$ expresses (the $\forall x P(x)$ is alternatively written as $\nexists x \neg P(x)$), as a result it holds that for $1 \leq i \leq n$, $\sigma_i \vDash_S \langle\!\langle f_{pr}\rangle\!\rangle$

Given the above, we get that $\sigma_1,\ldots,\sigma_n \in \varepsilon_{\|A\|}(premp)$ and
$\sigma_1(\langle\!\langle premp\rangle\!\rangle) \cup \cdots \cup \sigma_n(\langle\!\langle premp\rangle\!\rangle) = \|prem(a)\|$ and therefore, by the definition of $\mathcal{E}_G$:

$\|prem(a)\| \in \mathcal{E}_{\|A\|}(\langle\!\langle premp\rangle\!\rangle)$. Thus (1) holds.

2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle premp\rangle\!\rangle)$. By the definition of $\langle\!\langle premp\rangle\!\rangle$ g will be of the form

$g = \{(u(i)\ \textit{type I-node}\ ),\ (u(i)\ \textit{claimText val}),\ (u(i)\ \textit{premise}\ u(ra)),\ (u(ra)\ \textit{type RA-node})\ \}$

where $u(i)$ is the uri of an I-node instance, $u(ra)$ the uri of an RA-node instance, and $g \vDash \langle\!\langle f_{pr}\rangle\!\rangle$

According to table 4.1, the $\|A\|$, created by the translation of A can only contain i if there is a proposition p, where $u(\hat{p}) = u(i)$, for which it holds that $p \in prem(a)$, for some argument $a \in A$, for which $u(\hat{a}) = u(ra)$. Furthermore, since the $\langle\!\langle f_{pr}\rangle\!\rangle$ is satisfied, it means that there is some ra', such that $\forall i_x$ s.t. $(u(i_x)$ *premise* $u(ra'))$, there is also the triple $(u(i_x)$ *premise* $u(ra))$ or $(u(\hat{x'})$ *premise* $u(ra))$ given that there is a $df(i_x,x,i'_x,x')$ in $\|A\|$. Again here, $\|A\|$ can only contain ra' if $\exists a' \in A$, for which $u(\hat{a'}) = u(ra)$. For this a' it must hold that $\forall x \in prem(a')$, such that $u(\hat{x}) = u(i_x)$, it also holds that $x \in prem(a)$, or $\exists x'$, s.t. $x' \equiv x$ and $x' \in prem(a)$, for which $u(\hat{x'}) = u(i'_x)$. As a result $prem(a') \sqsubseteq prem(a)$.

Concluding, we infer that $prem(a) \in I_A(premp)$ and since $g \subseteq \|prem(a)\|$ we get that $g \subseteq \|I_A(premp)\|$. Thus (2) holds.

- If $premp = v\ [.\{sp_1,..,sp_m\}]$, with sp_j proposition pattern of propositions s_j.

  By the semantics of section 3.3.3, we have that:

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq P(A),\ \mu(v) = prem(a)\ \textit{for some}\ a \in A\ \textit{and}\ s_i \in prem(a),\ \textit{with}\ s_i \in I_A(sp_i), \textit{for some}\ 1 \leq i \leq m\ \}$

  Moreover, from table 4.2 (rules $r_8.2$ and $r_9.3$), we have that:

  $$\langle\!\langle \boldsymbol{f_{pr}}\rangle\!\rangle = \bigvee_{j=1}^{m} \Big[\ \big(\langle\!\langle sp_j\rangle\!\rangle \wedge (\langle\!\langle sp_j\rangle\!\rangle.w_i\ \textit{premise}\ w_{ra})\big)\ \ \vee\ \ \big(\ \langle\!\langle sp_j\rangle\!\rangle.\langle\!\langle eqp\rangle\!\rangle_{wi2=w_{ij}} \wedge (w_{ij}\ \textit{premise}\ w_{ra})\ \big)\Big]$$

  We decompose $\langle\!\langle f_{pr}\rangle\!\rangle$ into smaller graph patterns and rewrite it as follows:

$\langle\!\langle \textbf{\textit{premp}} \rangle\!\rangle = gp \wedge \langle\!\langle f_{pr} \rangle\!\rangle$ where

$gp = (w_i \textit{ type I-node}) \wedge (w_i \textit{ claimText } v_p) \wedge (w_i \textit{ premise } w_{ra}) \wedge (w_{ra} \textit{ type RA-node})$

$$\langle\!\langle f_{pr} \rangle\!\rangle = \bigvee_{j=1}^{m} gp_j$$

$gp_j = (\ gp_j^1 \textit{ UNION } gp_j^2\ )$

$gp_j^1 = \langle\!\langle sp_j \rangle\!\rangle \wedge (\langle\!\langle sp_j \rangle\!\rangle.w_i \textit{ premise } w_{ra})\ \big)$

$gp_j^2 = \langle\!\langle sp_j \rangle\!\rangle.\langle\!\langle eqp \rangle\!\rangle_{wi2=w_{ij}} \wedge (w_{ij} \textit{ premise } w_{ra})$

From the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(gp_j) = \sigma \in \varepsilon_A(gp_j^1) \textit{ or } \sigma \in \varepsilon_A(gp_j^2)$
- $\mathcal{E}_{\|A\|}(gp_j) = \mathcal{E}_{\|A\|}(gp_j^1) \cup \mathcal{E}_{\|A\|}(gp_j^2)$
- $\mathcal{E}_{\|A\|}(premp) = \{\{\sigma(gp) \cup \sigma(\langle\!\langle f_{pr} \rangle\!\rangle)\} \mid \sigma \in (\varepsilon_A(gp) \bowtie \varepsilon_A(gp_n)) \textit{ or} \dots \textit{or } \sigma \in (\varepsilon_A(gp) \bowtie \varepsilon_A(gp_n))$
$\}$

a) For the left to right direction, take a set of propositions $\{p_1, \dots, p_n\} \in I_A(premp)$. This means that $\exists a \in A : prem(a) = \{p_1, \dots, p_n\}$. In addition, $\exists s'_j \in I_A(sp_j)$ for some $1 \leq j \leq m$, such that $s'j \in prem(a)$. Without loss of generality, we have that $prem(a) = \{p_1, .., s'_j, .., p_n\}$.
From the table 4.1 we get that

$\|prem(a)\| = \{\ \|p_1\|, (u(\hat{p}_1) \textit{ premise } u(\hat{a})),\ ..\ , \|s'_j\|, (u(\hat{s'_j}) \textit{ premise } u(\hat{a})),\ \dots,\ \|p_n\|,\ (\ u(\hat{p}_n) \textit{ premise } u(\hat{a})), (u(\hat{a}) \textit{ type RA-node})\ \}$

Obviously, we have that $\|prem(a)\| \in \|I_A(premp)\|$, and also, recalling the definition of $\|A\|$, we have that $\|prem(a)\| \subseteq \|A\|$.

Let n different σ mappings of *premp*, such that:

$\sigma_1 = \{(w_i,\ u(\hat{p}_1)),\ (v,\ p_1),\ (w_{ra},\ u(\hat{a})),\ map_{\sigma_1}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )\}$

$\dots$

$\sigma_n = \{(w_i,\ u(\hat{p}_n)),\ (v,\ p_n),\ (w_{ra},\ u(\hat{a})),\ map_{\sigma_n}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )\}$

It holds that each $map_{\sigma_i}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )$ can be at least **one** of the following:

$map_{\sigma_i}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ ) = \big(map_{s1},\ (w_{ra}, u(\hat{a}))\big)$

or

...

or

$map_{\sigma_i}(\ \langle\langle f_{pr} \rangle\rangle\ ) = \big(map_{sm},\ (w_{ra}, u(\hat{a}))\big)$

where for $1 \leq j \leq m$, map_{sj} is the respective mapping for gp_j, depending on whether $\sigma_i \in \varepsilon_{\|KB\|}(gp_j^1)$ or $\sigma_i \in \varepsilon_{\|KB\|}(gp_j^2)$ (similar to the previous case where the premise pattern was a set of proposition patterns). In particular it holds that either $map_{sj} = (\langle\langle sp_j \rangle\rangle.w_i,\ u(\hat{s_j}))$ or $map_{sj} = \{\ (\langle\langle sp_j \rangle\rangle.w_i,\ u(\hat{s_j})),\ (\langle\langle sp_j \rangle\rangle.\langle\langle eqp \rangle\rangle.w_{i2},\ u(\hat{s'_j}))\ \}$.

Given the above, we get that $\sigma_1, \ldots, \sigma_n \in \varepsilon_{\|A\|}(premp)$ and
$\|prem(a)\| \subseteq \sigma_1(\langle\langle premp \rangle\rangle) \cup \cdots \cup \sigma_n(\langle\langle premp \rangle\rangle)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$:

$\|prem(a)\| \subseteq \mathcal{E}_{\|A\|}(\langle\langle premp \rangle\rangle)$. Thus (1) holds.

2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle premp \rangle\rangle)$. By the definition of $\langle\langle premp \rangle\rangle$ g will be of the form

$g = \{(u(i)\ \mathit{type\ I\text{-}node}\ ),\ (u(i)\ \mathit{claimText\ val}),\ (u(i)\ \mathit{premise}\ u(ra)),\ g_{f_{pr}}\}$

where $u(i)$ is the uri of an I-node instance, $u(ra)$ the uri of an RA-node instance, and
$g_{f_{pr}} = g_1\ or\ \ldots\ or\ g_m$

For each g_j, we discern two cases:

a) if $g_j \in \mathcal{E}_{\|A\|}(gp_j^1)$, then g will be of the form:

$g_j = \{\ (u(i_j)\ \mathit{type\ I\text{-}node}),\ (u(i_j)\ \mathit{claimText}\ s_j),\ (u(i_j)\ \mathit{premise}\ u(ra))\}$

where $u(i_j)$ is the uri of an I-node instance, and $u(ra)$ the uri of an RA-node instance. Moreover $g_j \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g_j if there is a proposition $s_j \in props(a)$, where $u(\hat{s_j}) = u(i_j)$, for which it holds that $s_j \in prem(a)$, for some argument $a \in A$, such that $u(\hat{a}) = u(ra)$.

b) Respectively, if $g_j \in \mathcal{E}_{\|A\|}(gp_j^2)$, then g_j will be of the form:

$g_j = \{\ (u(i_j)\ \mathit{type\ I\text{-}node}),\ (u(i_j)\ \mathit{claimText}\ s_j),\ \mathit{df}(i_j,\ s_j,\ i'_j,\ s'_j),\ (u(i'_j)\ \mathit{premise}\ u(ra))\}$

where $u(i_j), u(i'_j)$ are uris of two I-node instances i_j, i'_j, $u(ra)$ the uri of an RA-node instance and $\mathit{df}(i_j,\ s_j,\ i'_j,\ s'_j)$ is a default rephrase between i_j and i'_j. Moreover $g_j \subseteq \|A\|$.

$\|A\|$, created by the translation of A can only contain g if the following are satisfied by A:

- There is a proposition $s_j \in props(A)$, where $u(\hat{s_j}) = u(i_j)$

- There is an equivalence $e = (s_j, s'_j)$, such that $\|\hat{e}\| = df(i_j, s_j, i'_j, s'_j)$, $u(\hat{x'_j}) = u(i'_j)$ where $s'_j \in props(A)$ a second proposition
- For proposition s'_j, it also holds that $s'_j \in prem(a)$, for some $a \in A$, for which $u(\hat{a}) = u(ra)$.

As a result, we have that there is at least one proposition $s_j \in props(A)$ for which $s_j \in prem(a)$ or $s'_j \in prem(a)$, for $s_j \equiv s'_j$.

Concluding, we infer that $prem(a) \in I_A(premp)$ and since $g \subseteq \|prem(a)\|$ we get that $g \subseteq \|I_A(premp)\|$. Thus (2) holds.

- if $premp = v\ [.v']$,with $v' \in V$ a variable, appearing in the premise pattern of another argument pattern ap'.

  By the semantics of section 3.3.3, we have that:

  $I_A(premp) = \{\mu(v) \mid \mu(v) \subseteq P(A), \mu(v) = prem(a)$ for some $a \in A$, $\mu(v') = prem(a')$ for some $a' \in I_A(ap')$ and $\mu(v') \odot \mu(v)$ is true$\}$

  Moreover, from table 4.2 (rules $r_8.2$ and $r_9.4$), we have that:

  $\langle\!\langle \boldsymbol{premp} \rangle\!\rangle = (w_i\ type\ I\text{-}node) \wedge (w_i\ claimText\ v_p) \wedge (w_i\ premise\ w_{ra}) \wedge (w_{ra}\ type\ RA\text{-}node) \wedge \langle\!\langle f_{pr} \rangle\!\rangle_{w_{ra}}$

  $\langle\!\langle \boldsymbol{f_{pr}} \rangle\!\rangle = (\langle\!\langle premp_2 \rangle\!\rangle.w_i\ premise\ w_{ra}) \bigvee \Big(\langle\!\langle eqp \rangle\!\rangle_{wi1 = \langle\!\langle premp_2 \rangle\!\rangle.w_i} \wedge (\langle\!\langle eqp \rangle\!\rangle.w_{i2}\ premise\ w_{ra}) \Big)$

  We decompose $\langle\!\langle f_{pr} \rangle\!\rangle$ into smaller graph patterns and rewrite it as follows:

  $\langle\!\langle \boldsymbol{premp} \rangle\!\rangle = gp \wedge \langle\!\langle f_{pr} \rangle\!\rangle$

  where

  $gp = (w_i\ type\ I\text{-}node) \wedge (w_i\ claimText\ v_p) \wedge (w_i\ premise\ w_{ra}) \wedge (w_{ra}\ type\ RA\text{-}node)$

  $\langle\!\langle f_{pr} \rangle\!\rangle = gp_1\ UNION\ gp_2$

  $gp_1 = (\langle\!\langle premp_2 \rangle\!\rangle.w_i\ premise\ w_{ra})$

  $gp_2 = \langle\!\langle eqp \rangle\!\rangle_{wi1 = \langle\!\langle premp_2 \rangle\!\rangle.w_i} \wedge (\langle\!\langle eqp \rangle\!\rangle.w_{i2}\ premise\ w_{ra})$

  From the SPARQL semantics, we have the following:

  - $\varepsilon_A(\langle\!\langle f_{pr} \rangle\!\rangle) = \{\sigma \mid \sigma \in \varepsilon_A(gp_1)\ or\ \sigma \in \varepsilon_A(gp_2)\}$
  - $\varepsilon_A(\langle\!\langle premp \rangle\!\rangle) = \{\sigma \mid \sigma \in (\varepsilon_A(gp) \bowtie \varepsilon_A(\langle\!\langle f_{pr} \rangle\!\rangle))\}$
  - $\mathcal{E}_{\|A\|}(\langle\!\langle f_{pr} \rangle\!\rangle) = \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2)$

- $\mathcal{E}_{\|A\|}(\langle\!\langle premp \rangle\!\rangle) = \{\sigma(gp) \cup \sigma(\langle\!\langle f_{pr} \rangle\!\rangle) \mid \sigma \in \varepsilon_A(\langle\!\langle premp \rangle\!\rangle)\}$

a) For the left to right direction, take a set of propositions $\{p_1, \ldots, p_n\} \in I_A(premp)$. This means that $\exists a \in A : prem(a) = \{p_1, \ldots, p_n\}$. It also means that ap' has been evaluated and $\exists a' \in I_A(ap')$, s.t. $\mu(v') = prem(a')$ and $prem(a') \odot prem(a)$ is true (here we have $\odot$ instead of $\cap$). In other words, $\exists x \in prem(a')$, such that $x \in prem(a)$ or $\exists x' : x \equiv x'$ and $x' \in prem(a')$.

From the table 4.1, we get that

$\|prem(a)\| = \{ \|p_1\|, (u(\hat{p_1})\ premise\ u(\hat{a})), \ldots, \|x\|, (u(\hat{x})\ premise\ u(\hat{a})), \ldots, \|p_n\|, (u(\hat{p_n})\ premise\ u(\hat{a})), (u(\hat{a})\ type\ RA\text{-}node)\}$

Obviously, we have that $\|prem(a)\| \in \|I_A(premp)\|$, and recalling the definition of $\|A\|$, we also have that $\|prem(a)\| \subseteq \|A\|$.

Let n different σ mappings of $\langle\!\langle premp \rangle\!\rangle$, such that:

$\sigma_1 = \{(w_i,\ u(\hat{p_1}),\ (v,\ pval_1),\ (w_{ra},\ u(\hat{a})),\ map_{\sigma_1}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )\}$

$\ldots$

$\sigma_n = \{(w_i,\ u(\hat{p_n}),\ (v,\ pval_n),\ (w_{ra},\ u(\hat{a})),\ map_{\sigma_n}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ )\}$

where $1 \leq i \leq n$, $pval_i$ is the text value of proposition p_i and

$map_{\sigma_i}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ ) = \{(\langle\!\langle premp_2 \rangle\!\rangle.w_i,\ u(\hat{x})), (w_{ra}, u(\hat{a}))\}$ or

$map_{\sigma_i}(\ \langle\!\langle f_{pr} \rangle\!\rangle\ ) = \{(\langle\!\langle premp_2 \rangle\!\rangle.w_i,\ u(\hat{x'})), (\langle\!\langle eqp \rangle\!\rangle.w_{i2},\ u(\hat{x})), (w_{ra}, u(\hat{a}))\}$

depending on whether $\sigma_i \in \varepsilon_{\|KB\|}(gp_1)$ or $\sigma_i \in \varepsilon_{\|KB\|}(gp_2)$.

Given the above, we get that $\sigma_1, \ldots, \sigma_n \in \varepsilon_{\|A\|}(premp)$ and
$\|prem(a)\| \subseteq \sigma_1(\langle\!\langle premp \rangle\!\rangle) \cup \cdots \cup \sigma_n(\langle\!\langle premp \rangle\!\rangle)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$:

$\|prem(a)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle premp \rangle\!\rangle)$. Thus (1) holds.

2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle premp \rangle\!\rangle)$. By the definition of $\langle\!\langle premp \rangle\!\rangle$, g will be of the form

$g = \{(u(i)\ type\ I\text{-}node\ ),\ (u(i)\ claimText\ p),\ (u(i)\ premise\ u(ra)),\ g_{f_{pr}}\}$

where $u(i)$ is the uri of an I-node instance, $u(ra)$ the uri of an RA-node instance. Regarding $g_{f_{pr}}$ we discern two cases:

a) If $g_{f_{pr}} \in \mathcal{E}_{\|A\|}(gp_1)$, then it will have the form:

$g_{f_{pr}}$ = { $(u(i_x)$ *type I-node*), $(u(i_x)$ *claimText x*), $(u(i_x)$ *premise* $u(ra))$}

where $u(i_x)$ is the uri of an I-node instance, for which, we have $\sigma(\langle\!\langle premp_2 \rangle\!\rangle.w_i) = u(i_x)$ and $u(ra)$ the uri of the RA-node instance above. It will also hold that, $((u(i_x)\ premise\ u(ra')) \in \mathcal{E}_{\|A\|}(\langle\!\langle premp_2 \rangle\!\rangle)$. Moreover $g \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain i_x if there is a proposition $x \in props(A)$, where $u(\hat{x}) = u(i_x)$, for which it holds that $x \in prem(a)$, for some argument $a \in A$, such that $u(\hat{a}) = u(ra)$ and also $x \in prem(a')$, for some $a' \in A$, such that $u(\hat{a'}) = u(ra')$.

b) Respectively, if $g_{f_{pr}} \in \mathcal{E}_{\|A\|}(gp_1)$, then it will have the form:

$g_{f_{pr}}$ = { $(u(i_x)$ *type I-node*), $(u(i_x)$ *claimText x*), *df*(i_x, x, i'_x, x'), $(u(i'_x)$ *premise* $u(ra))$}

where $u(i_x), u(i'_x)$ are uris of two I-node instances i_x, i'_x, $u(ra)$ the uri of an RA-node instance and *df*(i_x, x, i'_x, x') is a default rephrase between i_x and i'_x. It will also hold that, $((u(i'_x)\ premise\ u(ra')) \in \mathcal{E}_{\|A\|}(\langle\!\langle premp_2 \rangle\!\rangle)$. Moreover $g \subseteq \|A\|$.

$\|A\|$, created by the translation of A can only contain g if the following are satisfied by A:

- There is a proposition $x \in props(A)$, where $u(\hat{x}) = u(i_x)$ and value val_x
- There is an equivalence $e = (x, x')$, such that $\|\hat{e}\| = df(i_x, val_x, i'_x, val'_x)$, $u(\hat{x'}) = u(i'_x)$ where $x' \in props(A)$ a second proposition with value val'_x.
- For proposition x', it holds that $x' \in prem(a)$, for some $a \in A$, for which $u(\hat{a}) = u(ra)$ and also $x' \in prem(a')$, for some $a' \in A$, for which $u(\hat{a'}) = u(ra')$.

As a result, we have that $prem(a) \sqcap prem(a')$, which means that $prem(a) \in I_A(premp)$ and $prem(a') \in I_A(premp_2)$ and since $g \subseteq \|prem(a)\|$ we get that $g \subseteq \|I_A(premp)\|$. Thus (2) holds.

Argument pattern

We now move to the case of an argument pattern $ap = \langle\ premp,\ conclp\ \rangle$, where *premp* and *conclp*, premise and conclusion patterns respectively. By the semantics of section 3.3.3 we have that:

$I_A(ap) = \{a \in A \mid prem(a) \in I_A(premp)$ and $concl(a) \in I_A(conclp)\}$

Moreover, from table 4.3 (rule r_{11}), we have that:

$\langle\!\langle \boldsymbol{ap} \rangle\!\rangle = \langle\!\langle premp \rangle\!\rangle_{w_{ra}} \quad \wedge \quad \langle\!\langle conclp \rangle\!\rangle_{w_{ra}} \quad \wedge \quad (w_{ra}\ type\ RA\text{-}node\ )$

(The case where an argument pattern is a single variable will not be examined separately, since it is equivalent with the case $\langle ?pr, ?c \rangle$, (no filters in the premise pattern) so the proof for these two cases remains the same.)

(1) For the left to right direction, take an argument $a \in I_A(ap)$. This means that $prem(a) \in I_A(premp)$ and $concl(a) \in I_A(conclp)$. According to the table 4.1, we have that:

$\|a\| = \{\|prem(a)\| \cup \|concl(a)\| \cup \{(u(\hat{a})$ type RA-node$)\}\}$

Obviously, given that $\|I_A(ap)\| = \{\|a_i\| \mid a_i \in I_A(ap)\}$, we have that $\|a\| \in \|I_A(ap)\|$ and that $\|a\| \subseteq \|A\|$. By the proofs of the premise and conclusion patterns, we showed that, regardless their particular form, it holds that $\|prem(a)\| \in \mathcal{E}_{\|A\|}(\langle\!\langle premp\rangle\!\rangle)$ and $\|concl(a)\| \in \mathcal{E}_{\|A\|}(\langle\!\langle conclp\rangle\!\rangle)$.

Given the above, for $\sigma = \{w_{ra}, u(\hat{a})\}$, we get that $\sigma \in \varepsilon_{\|A\|}(\langle\!\langle ap\rangle\!\rangle)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|a\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle ap\rangle\!\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle ap\rangle\!\rangle)$. By the definition of $\langle\!\langle ap\rangle\!\rangle$ given above, we get that g will be of the form:

$g = \{g_1 \cup g_2 \cup \{(u(ra)\ type\ RA\text{-}node\ )\}\}$

where u_{ra} is a uri of a RA instance, $g_1 \in I_A(\langle\!\langle premp\rangle\!\rangle)$ and $g_2 \in I_A(\langle\!\langle conclp\rangle\!\rangle)$ Moreover $g \subseteq \|A\|$.

According to table 4.1, the $\|A\|$, created by the translation of A can only contain g if there is an argument $a \in A$, where $u(\hat{a}) = u(ra)$, for which $prem(a) \in I_A(premp)$ (by the proof of premise pattern) and $concl(a) \in I_A(conclp)$ (by the proof of premise pattern). We note that $\|\hat{a}\| = g$, therefore $g \in \|I_A(\langle\!\langle ap\rangle\!\rangle)\|$ and thus (2) holds.

Relation pattern

We assume a relation pattern, *relp*. We have to show that for each of the six different types of *relp* (rebut, undercut, attack, endorse, back, support), it holds that $\|I_A(relp)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle relp\rangle\!\rangle)$(1) and $\mathcal{E}_{\|A\|}(\langle\!\langle relp\rangle\!\rangle) \in \|I_A(relp)\|$(2). Recall that each relation pattern appears between two **random** argument patterns $ap_1 =< premp_1, cp_1 >$ and $ap_2 =< premp_2, cp_2 >$, with $premp_1$, $premp_2$ premise patterns and cp_1, cp_2 conclusion patterns, respectively. So we have:

- **(*Rebut*)** By the semantics of section 3.3.3, we have that:

  $I_A(rebut) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A$ s.t. concl$(a_1) \not\vdash$ concl$(a_2)\ \}$

  Moreover, from table 4.3 (rule $(r_{12}.1)$), we have that:

  $$\begin{aligned}\langle\!\langle \textbf{\textit{rebut}}\rangle\!\rangle = \langle\!\langle ap_1\rangle\!\rangle \wedge \langle\!\langle ap_2\rangle\!\rangle \wedge \Big[\ \big(\ \langle\!\langle conflp\rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1\rangle\!\rangle.w_i,\ w_{i2}=\langle\!\langle cp_2\rangle\!\rangle.w_i}\ \big) \vee \\ \big(\ \langle\!\langle eqp\rangle\!\rangle_{w_{i1}=\langle\!\langle cp_2\rangle\!\rangle.w_i} \wedge \langle\!\langle conflp\rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1\rangle\!\rangle.w_i,\ w_{i2}=\langle\!\langle eqp\rangle\!\rangle.w_{i2}}\ \big) \vee \\ \big(\ \langle\!\langle eqp\rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1\rangle\!\rangle.w_i} \wedge \langle\!\langle conflp\rangle\!\rangle_{w_{i1}=\langle\!\langle eqp\rangle\!\rangle.w_{i2},\ w_{i2}=\langle\!\langle cp_2\rangle\!\rangle.w_i}\ \big) \vee \\ \big(\ \langle\!\langle eqp_1\rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1\rangle\!\rangle.w_i} \wedge \langle\!\langle eqp_2\rangle\!\rangle_{w_{i1}=\langle\!\langle cp_2\rangle\!\rangle.w_i} \wedge \langle\!\langle conflp\rangle\!\rangle_{w_{i1}=\langle\!\langle eqp_1\rangle\!\rangle.w_{i2},\ w_{i2}=\langle\!\langle eqp_2\rangle\!\rangle.w_{i2}}\ \big)\ \Big]\end{aligned}$$

  We decompose and rewrite $\langle\!\langle rebut\rangle\!\rangle$ into smaller graph patterns and rewrite it as:

$\langle\!\langle rebut \rangle\!\rangle = gp_1$ *UNION* gp_2 *UNION* gp_3 *UNION* gp_4 such that

$gp_1 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i,\, w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i}$

$gp_2 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i,\, w_{i2}=\langle\!\langle eqp \rangle\!\rangle.w_{i2}}$

$gp_3 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle eqp \rangle\!\rangle.w_{i2},\, w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i}$

$gp_4 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \langle\!\langle eqp_1 \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i} \wedge \langle\!\langle eqp_2 \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \wedge \langle\!\langle conflp \rangle\!\rangle_{w_{i1}=\langle\!\langle eqp_1 \rangle\!\rangle.w_{i2},\, w_{i2}=\langle\!\langle eqp_2 \rangle\!\rangle.w_{i2}}$

By the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(rebut) = \sigma \in \varepsilon_A(gp_1)$ *or* $\sigma \in \varepsilon_A(gp_2)$ *or* $\sigma \in \varepsilon_A(gp_3)$ *or* $\sigma \in \varepsilon_A(gp_4)$

- $\mathcal{E}_{\|A\|}(rebut) = \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2) \cup \mathcal{E}_{\|A\|}(gp_3) \cup \mathcal{E}_{\|A\|}(gp_4)$

(1) For the left to right direction take a relation $r \in I_A(rebut)$. This means that $r = (a_1\ a_2)$, where $a_1, a_2 \in A$. Analyzing he proposition 2 of section 3.1, we infer that there is a rebut relation between a_1 and a_2 in one of the four cases bellow:

a) $concl(a_1) \not\to concl(a_2)$
b) $concl(a_1) \not\to x$ and $x \equiv concl(a_2)$
c) $concl(a_1) \equiv x$ and $x \not\to concl(a_2)$
d) $concl(a_1) \equiv x$ and $concl(a_2) \equiv y$ and $x \not\to y$

According to the table 4.1, in each of these cases, r is translated as follows:

a) $\|r\| = \{\|a_1\|, \|a_2\|, \|c\|\}$, where $c \in cf(A)$ and $c = (concl(a_1), concl(a_2))$.

b) $\|r\| = \{\|a_1\|, \|a_2\|, \|c\|, \|e\|\}$, where $c \in cf(A)$ with $c = (concl(a_1), x)$ and $e \in eq(A)$ with $e = (x, concl(a_2))$.

c) $\|r\| = \{\|a_1\|, \|a_2\|, \|e\|, \|c\|\}$, where $e \in eq(A)$ with $e = (concl(a_1), x)$ and $c \in cf(A)$ with $c = (x, concl(a_2))$.

d) $\|r\| = \{\|a_1\|, \|a_2\|, \|e_1\|, \|e_2\|, \|c\|\}$, where $e_1, e_2 \in eq(A)$ with $e_1 = (concl(a_1), x)$, $e_2 = (concl(a_2), y)$ and $c \in cf(A)$ with $c = (x, y)$.

Obviously, given that $\|I_A(rebut)\| = \{\ \|r_i\| \mid r_i \in I_A(r_i)\ \}$, we have that $\|r\| \in \|I_A(rebut)\|$ and that $\|r\| \subseteq \|A\|$. Furthermore, by the proof of an argument pattern above and given that ap_1, ap_2 are random argument patterns which match any argument, we have that $\|a_1\| \in \mathcal{E}_{\|A\|}(\langle\!\langle ap_1 \rangle\!\rangle)$ and $\|a_2\| \in \mathcal{E}_{\|A\|}(\langle\!\langle ap_2 \rangle\!\rangle)$

a) If r belongs to the case (a), then for $\sigma = \{(\langle\!\langle ap_1 \rangle\!\rangle.w_{ra},\ u(\hat{a}_1)), (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}, u(\hat{a}_2)), (\langle\!\langle conflp \rangle\!\rangle.w_{i1}, u(\hat{c}_1)), (\langle\!\langle conflp \rangle\!\rangle.w_{i2}, u(\hat{c}_2)), (\langle\!\langle conflp \rangle\!\rangle.w_{ca}, u(\hat{c}))\ \}$, where $c_1 = concl(a_1)$ and $c_2 = concl(a_2)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_1)$ which generates a $ca(\hat{c_1},\hat{c_2})$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_1)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle rebut \rangle\!\rangle)$. Thus (1) holds.

b) If r belongs to the case (b), then for

$\sigma = \{((\langle\!\langle ap_1 \rangle\!\rangle.w_{ra},\ u(\hat{a_1})), (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}, u(\hat{a_2})),(\langle\!\langle conflp \rangle\!\rangle.w_{i1}, u(\hat{c_1})), (\langle\!\langle conflp \rangle\!\rangle.w_{i2}, u(\hat{x})), (\langle\!\langle eqp \rangle\!\rangle.w_{i1}, u(\hat{c_2})), (\langle\!\langle eqp \rangle\!\rangle.w_{i2}, u(x)), (\langle\!\langle conflp \rangle\!\rangle.w_{ca}, u(\hat{c})), (\langle\!\langle eqp \rangle\!\rangle.w_{ma}, u(\hat{e}))\ \}$

where $c_1 = concl(a_1)$ and $c_2 = concl(a_2)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_2)$, which generates a $ca(\hat{x},\hat{c_1})$ and a $df(\hat{c_2},c_2,\hat{x},x)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_2)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle rebut \rangle\!\rangle)$. Thus (1) holds.

c) If r belongs to the case (c), then for

$\sigma = \{((\langle\!\langle ap_1 \rangle\!\rangle.w_{ra},\ u(\hat{a_1})), (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}, u(\hat{a_2})),(\langle\!\langle eqp \rangle\!\rangle.w_{i1}, u(\hat{c_1})), (\langle\!\langle eqp \rangle\!\rangle.w_{i2}, u(x)), (\langle\!\langle conflp \rangle\!\rangle.w_{i1}, u(\hat{c_2})), (\langle\!\langle conflp \rangle\!\rangle.w_{i2}, u(\hat{x})), (\langle\!\langle conflp \rangle\!\rangle.w_{ca}, u(\hat{c})), (\langle\!\langle eqp \rangle\!\rangle.w_{ma}, u(\hat{e}))\ \}$

where $c_1 = concl(a_1)$ and $c_2 = concl(a_2)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_3)$, which generates a $ca(\hat{x},\hat{c_1})$ and a $df(\hat{c_2},c_2,\hat{x},x)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_3)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle rebut \rangle\!\rangle)$. Thus (1) holds.

d) If r belongs to the case (d), then for

$\sigma = \{((\langle\!\langle ap_1 \rangle\!\rangle.w_{ra},\ u(\hat{a_1})),\ (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}, u(\hat{a_2})),\ (\langle\!\langle eqp_1 \rangle\!\rangle.w_{i1}, u(\hat{c_1})),\ (\langle\!\langle eqp_1 \rangle\!\rangle.w_{i2}, u(\hat{x})),\ (\langle\!\langle eqp_2 \rangle\!\rangle.w_{i1}, u(\hat{c_2})),\ (\langle\!\langle eqp_2 \rangle\!\rangle.w_{i2}, u(\hat{y})),\ (\langle\!\langle conflp \rangle\!\rangle.w_{i1}, u(\hat{x})),\ (\langle\!\langle conflp \rangle\!\rangle.w_{i2}, u(\hat{y})),\ (\langle\!\langle eqp_1 \rangle\!\rangle.w_{ma}, u(\hat{e_1})),\ (\langle\!\langle eqp_2 \rangle\!\rangle.w_{ma}, u(\hat{e_2})),\ (\langle\!\langle conflp \rangle\!\rangle.w_{ca}, u(\hat{c}))\ \}$

where $c_1 = concl(a_1)$ and $c_2 = concl(a_2)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_4)$, which generates a $df(\hat{c_1},c_1,\hat{x},c_x)$, a $df(\hat{c_2},c_2,\hat{y},y)$ and a $ca(\hat{x},\hat{y})$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_4)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle rebut \rangle\!\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle rebut \rangle\!\rangle)$. By the definition of $\langle\!\langle rebut \rangle\!\rangle$ we have one of the four cases:

a) If $g \in \mathcal{E}_{\|A\|}(\langle\!\langle gp_1 \rangle\!\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ ca(i_1,i_2)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\!\langle ap_1 \rangle\!\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\!\langle ap_2 \rangle\!\rangle)$ and $ca(i_1,i_2)$ a conflict application between two I-nodes i_1,i_2. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1,a_2 \in A$. In addition, given that the conflict pattern is between the variables $\langle\!\langle cp_1 \rangle\!\rangle.w_i$ and $\langle\!\langle cp_2 \rangle\!\rangle.w_i$, the $\|A\|$, created by the translation of A, can only contain $ca(i_1,i_2)$ if $i_1 = concl(a_1)$, $i_2 = concl(a_2)$ and $concl(a_1) \not\sim concl(a_2)$. Thus, we infer that there is a rebut relation r between a_1 and a_2 (case a). We note that $g = \|r\|$, and therefore, $g \in \|I_A(rebut)\|$. Thus (2) holds.

b) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_2 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1,c_1,i_2,c_2),\ ca(i_2,i_3)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$, $ca(i_1,i_2)$ a *conflict application* between two I-nodes i_1,i_2 and $df(i_1,c,i_2,x)$ a *default rephrase* between the I-nodes i_2,i_3. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1,a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern is $\langle\langle cp_2 \rangle\rangle.w_i$ and that one of the two variables of the conflict pattern is the $\langle\langle cp_1 \rangle\rangle.w_i$, and that there is a join between them, we infer the following:

- $\|A\|$, created by the translation of A, can only contain $df(i_1,c,i_2,x)$ if $i_1 = \hat{c}$ with $c = concl(a_2)$, $i_2 = \hat{x}$ and $c \equiv x$ for $c,x \in \mathcal{P}$.
- $\|A\|$ can only contain $ca(i_2,i_3)$ if $i_2 = \hat{x}$, $i_3 = \widehat{concl}(a_1)$ and $concl(a_1) \not\equiv x$

Thus, we infer that there is a rebut relation r between a_1 and a_2 (case b). We note that $g = \|r\|$, and therefore, $g \in \|I_A(rebut)\|$. Thus (2) holds.

c) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_3 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1,val_1,i_2,val_2),\ ca(i_2,i_3)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$, $ca(i_1,i_2)$ a *conflict application* between two I-nodes i_1,i_2 and $df(i_2,c,i_3,x)$ a *default rephrase* between the I-nodes i_2,i_3. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1,a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern is $\langle\langle cp_1 \rangle\rangle.w_i$ and that one of the two variables of the conflict pattern is the $\langle\langle cp_2 \rangle\rangle.w_i$, and that there is a join between them, we infer the following:

- $\|A\|$ can only contain $ca(i_1,i_2)$ if $i_1 = \widehat{concl}(a_2)$, $i_2 = \hat{x}$, $concl(a_2) \not\equiv x$, for $x \in \mathcal{P}$.
- $\|A\|$, created by the translation of A, can only contain $df(i_2,x,i_3,c)$ if $i_2 = \hat{x}$, $i_3 = \hat{c}$ $c = concl(a_1)$ and $c \equiv x$, for $c,x \in \mathcal{P}$.

Thus, we infer that there is a rebut relation r between a_1 and a_2 (case c). We note that $g = \|r\|$, and therefore, $g \in \|I_A(rebut)\|$. Thus (2) holds.

d) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_4 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1,c_1,i_2,x),\ df(i_3,c_2,i_4,y),\ ca(i_2,i_4)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$, $df(i_1,c_1,i_2,x)$ a *default rephrase* between the I-nodes i_1,i_2, $df(i_3,c_2,i_4,y)$ a *default rephrase* between the I-nodes i_3,i_4 and $ca(i_2,i_4)$ a *conflict application* between the i_2,i_4. We showed from the previous case (argument pattern) that $g_1 \in$

$\|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern eqp_1 is $\langle\langle cp_1\rangle\rangle.w_i$, that one of the two variables in the equivalence pattern eqp_2 is $\langle\langle cp_2\rangle\rangle.w_i$ and that the conflict pattern joins with the variables $\langle\langle eqp_1\rangle\rangle.w_{i2}$ and $\langle\langle eqp_2\rangle\rangle.w_{i2}$, we infer the following:

- $\|A\|$, created by the translation of A, can only contain $df(i_1, c_1, i_2, x)$ if $i_1 = \hat{c_1}$, $c_1 = concl(a_1)$, $i_2 = \hat{x}$ and $c_1 \equiv x$ for $x, c_1 \in props(A)$
- $\|A\|$ can only contain $df(i_3, c_2, i_4, y)$ if $i_3 = \hat{c_2}$ $c_2 = concl(a_2)$, $i_4 = y$ and $c_2 \equiv y$ for $c_2, y \in props(A)$
- $\|A\|$ can only contain $ca(i_2, i_4)$ if $i_2 = \hat{x}$, $i_4 = \hat{y}$ and $x \not\sim y$

Thus, we infer that there is a rebut relation r between a_1 and a_2 (case d). We note that $g = \|r\|$, and therefore, $g \in \|I_A(rebut)\|$. Thus (2) holds.

- **(*Undercut*)** By the semantics of section 3.3.3, we have that:

$I_A(undercut) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A$ s.t. concl(a_1) $\not\sim p_i$ for some $p_i \in$ prem(a_2)$\}$

Moreover, from table 4.3 (rule $(r_{12}.2)$), we have that:

$$\begin{aligned}\langle\langle \textbf{\textit{undercut}}\rangle\rangle = \langle\langle ap_1\rangle\rangle \wedge \langle\langle ap_2\rangle\rangle \wedge \Big[& \big(\langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i,\ w_{i2}=\langle\langle premp_2\rangle\rangle.w_i} \big) \vee \\ & \big(\langle\langle eqp\rangle\rangle_{w_{i1}=\langle\langle premp_2\rangle\rangle.w_i} \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i,\ w_{i2}=\langle\langle eqp\rangle\rangle.w_{i2}} \big) \vee \\ & \big(\langle\langle eqp\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i} \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle eqp\rangle\rangle.w_{i2},\ w_{i2}=\langle\langle premp_2\rangle\rangle.w_i} \big) \vee \\ & \big(\langle\langle eqp_1\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i} \wedge \langle\langle eqp_2\rangle\rangle_{w_{i1}=\langle\langle premp_2\rangle\rangle.w_i} \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle eqp_1\rangle\rangle.w_{i2},\ w_{i2}=\langle\langle eqp_2\rangle\rangle.w_{i2}} \big) \Big]\end{aligned}$$

We decompose and rewrite $\langle\langle undercut\rangle\rangle$ into smaller graph patterns as follows:

$\langle\langle undercut\rangle\rangle = gp_1\ UNION\ gp_2\ UNION\ gp_3\ UNION\ gp_4$

such that

$gp_1 = \langle\langle ap_1\rangle\rangle \wedge \langle\langle ap_2\rangle\rangle \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i,\ w_{i2}=\langle\langle premp_2\rangle\rangle.w_i}$

$gp_2 = \langle\langle ap_1\rangle\rangle \wedge \langle\langle ap_2\rangle\rangle \wedge \langle\langle eqp\rangle\rangle_{w_{i1}=\langle\langle premp_2\rangle\rangle.w_i} \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i,\ w_{i2}=\langle\langle eqp\rangle\rangle.w_{i2}}$

$gp_3 = \langle\langle ap_1\rangle\rangle \wedge \langle\langle ap_2\rangle\rangle \wedge \langle\langle eqp\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i} \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle eqp\rangle\rangle.w_{i2},\ w_{i2}=\langle\langle premp_2\rangle\rangle.w_i}$

$gp_4 = \langle\langle ap_1\rangle\rangle \wedge \langle\langle ap_2\rangle\rangle \wedge \langle\langle eqp_1\rangle\rangle_{w_{i1}=\langle\langle cp_1\rangle\rangle.w_i} \wedge \langle\langle eqp_2\rangle\rangle_{w_{i1}=\langle\langle premp_2\rangle\rangle.w_i} \wedge \langle\langle conflp\rangle\rangle_{w_{i1}=\langle\langle eqp_1\rangle\rangle.w_{i2},\ w_{i2}=\langle\langle eqp_2\rangle\rangle.w_{i2}}$

From the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(undercut) = \sigma \in \varepsilon_A(gp_1)\ or\ \sigma \in \varepsilon_A(gp_2)\ or\ \sigma \in \varepsilon_A(gp_3)\ or\ \sigma \in \varepsilon_A(gp_4)$

- $\mathcal{E}_{\|A\|}(undercut) = \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2) \cup \mathcal{E}_{\|A\|}(gp_3) \cup \mathcal{E}_{\|A\|}(gp_4)$

(1) For the left to right direction, we assume $r \in I_A(undercut)$. This means that $r = P_{a_1 \to a_2} = (a_1\ a_2)$, where $a_1, a_2 \in A$. Taking into account the proposition 2 of section 3.1, we infer that there is an undercut relation between a_1 and a_2 in one of the four cases bellow. We assume some $p \in prem(a_2)$, then:

a) $concl(a_1) \not\prec p$
b) $concl(a_1) \not\prec x$ and $x \equiv p$
c) $concl(a_1) \equiv x$ and $x \not\prec p$
d) $concl(a_1) \equiv x$ and $p \equiv y$ and $x \not\prec y$

According to the data mapping of table 4.1 we have for each of these cases that:

a) $\|r\| = \{\|a_1\|, \|a_2\|, \|c\|\}$, where $c \in cf(A)$ and $c = (concl(a_1), p)$.

b) $\|r\| = \{\|a_1\|, \|a_2\|, \|c\|, \|e\|\}$, where $c \in cf(A)$ with $c = (concl(a_1), x)$ and $e \in eq(A)$ with $e = (x, p)$

c) $\|r\| = \{\|a_1\|, \|a_2\|, \|e\|, \|c\|\}$, where $e \in eq(A)$ with $e = (concl(a_1), x)$ and $c \in cf(A)$ with $c = (x, p)$

d) $\|r\| = \{\|a_1\|, \|a_2\|, \|e_1\|, \|e_2\|, \|c\|\}$, where $e_1, e_2 \in eq(A)$ with $e_1 = (concl(a_1), x)$, $e_2 = (p, y)$ and $c \in cf(A)$ with $c = (x, y)$.

Obviously, given that $\|I_A(undercut)\| = \{\ \|r_i\| \mid r_i \in I_A(r_i)\ \}$, we have that $\|r\| \in \|I_A(undercut)\|$ and that $\|r\| \subseteq \|A\|$. Furthermore, by the proof of an argument pattern above and given that ap_1, ap_2 are random argument patterns which match any argument, we have that $\|a_1\| \in \mathcal{E}_{\|A\|}(\langle\!\langle ap_1 \rangle\!\rangle)$ and $\|a_2\| \in \mathcal{E}_{\|A\|}(\langle\!\langle ap_2 \rangle\!\rangle)$

a) If r belongs to the case (a), then for $\sigma = \{(\langle\!\langle ap_1 \rangle\!\rangle.w_{ra},\ u(\hat{a_1})), (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}, u(\hat{a_2})), (\langle\!\langle conflp \rangle\!\rangle.w_{i1},\ u(\hat{c_1})), (\langle\!\langle conflp \rangle\!\rangle.w_{i2},\ u(\hat{p})), (\langle\!\langle conflp \rangle\!\rangle.w_{ca},\ u(\hat{c}))\ \}$, where $c_1 = concl(a_1)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_1)$ which generates a $ca(\hat{c_1}, \hat{p})$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_1)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle undercut \rangle\!\rangle)$. Thus (1) holds.

b) If r belongs to the case (b), then for

$\sigma = \{(\langle\!\langle ap_1 \rangle\!\rangle.w_{ra},\ u(\hat{a_1})), (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}, u(\hat{a_2})),(\langle\!\langle conflp \rangle\!\rangle.w_{i1}, u(\hat{c_1})), (\langle\!\langle conflp \rangle\!\rangle.w_{i2}, u(\hat{x})), (\langle\!\langle eqp \rangle\!\rangle.w_{i1},\ u(\hat{p})), (\langle\!\langle eqp \rangle\!\rangle.w_{i2},\ u(x)), (\langle\!\langle conflp \rangle\!\rangle.w_{ca},\ u(\hat{c})), (\langle\!\langle eqp \rangle\!\rangle.w_{ma},\ u(\hat{e}))\ \}$

where $c_1 = concl(a_1)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_2)$, which generates a $ca(\hat{x}, \hat{c_1})$ and a $df(\hat{p}, p, \hat{x}, x)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_2)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle undercut \rangle\!\rangle)$. Thus (1) holds.

c) If r belongs to the case (c), then for

$\sigma = \{ (\langle\langle ap_1 \rangle\rangle.w_{ra},\ u(\hat{a_1})), (\langle\langle ap_2 \rangle\rangle.w_{ra}, u(\hat{a_2})), (\langle\langle eqp \rangle\rangle.w_{i1}, u(\hat{c_1})), (\langle\langle eqp \rangle\rangle.w_{i2}, u(x)), (\langle\langle conflp \rangle\rangle.w_{i1},$ $u(\hat{p})), (\langle\langle conflp \rangle\rangle.w_{i2}, u(\hat{x})), (\langle\langle conflp \rangle\rangle.w_{ca}, u(\hat{c})), (\langle\langle eqp \rangle\rangle.w_{ma}, u(\hat{e}))\ \}$

where $c_1 = concl(a_1)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_3)$, which generates a $ca(\hat{x}, \hat{c_1})$ and a $df(\hat{p}, p, \hat{x}, x)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_3)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\langle rebut \rangle\rangle)$. Thus (1) holds.

d) If r belongs to the case (d), then for

$\sigma = \{ (\langle\langle ap_1 \rangle\rangle.w_{ra},\ u(\hat{a_1})),\ (\langle\langle ap_2 \rangle\rangle.w_{ra}, u(\hat{a_2})),\ (\langle\langle eqp_1 \rangle\rangle.w_{i1}, u(\hat{c_1})),\ (\langle\langle eqp_1 \rangle\rangle.w_{i2}, u(\hat{x})),$ $(\langle\langle eqp_2 \rangle\rangle.w_{i1}, u(\hat{p})),\ (\langle\langle eqp_2 \rangle\rangle.w_{i2}, u(\hat{y})),\ (\langle\langle conflp \rangle\rangle.w_{i1}, u(\hat{x})),\ (\langle\langle conflp \rangle\rangle.w_{i2}, u(\hat{y})),$ $(\langle\langle eqp_1 \rangle\rangle.w_{ma}, u(\hat{e_1})),\ (\langle\langle eqp_2 \rangle\rangle.w_{ma}, u(\hat{e_2})),\ (\langle\langle conflp \rangle\rangle.w_{ca}, u(\hat{c}))\ \}$

where $c_1 = concl(a_1)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_4)$, which generates a $df(\hat{c_1}, c_1, \hat{x}, x)$, a $df(\hat{c_2}, c_2, \hat{y}, y)$ and a $ca(\hat{x}, \hat{y})$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_4)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\langle undercut \rangle\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle undercut \rangle\rangle)$. By the definition of $\langle\langle undercut \rangle\rangle$ we have one of the four cases:

a) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_1 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ ca(i_1, i_2)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$ and $ca(i_1, i_2)$ a conflict application between two I-nodes i_1, i_2. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. In addition, given that the conflict pattern is between the variables $\langle\langle cp_1 \rangle\rangle.w_i$ and $\langle\langle premp_2 \rangle\rangle.w_i$, the $\|A\|$, created by the translation of A, can only contain $ca(i_1, i_2)$ if $i_1 = concl(a_1)$, $i_2 = p$, for $p \in prem(a_2)$ and $concl(a_1) \not\sim p$. Thus, we infer that there is an undercut relation r between a_1 and a_2 (case a). We note that $g = \|r\|$, and therefore, $g \in \|I_A(udnercut)\|$. Thus (2) holds.

b) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_2 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1, p_1, i_2, x),\ ca(i_2, i_3)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$, $ca(i_1, i_2)$ a *conflict application* between two I-nodes i_1, i_2 and $df(i_1, p_1, i_2, x)$ a *default rephrase* between the I-nodes i_2, i_3. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern is $\langle\langle premp_2 \rangle\rangle.w_i$ and that one of the two variables of the conflict pattern is the $\langle\langle cp_1 \rangle\rangle.w_i$, and that there is a join between them, we infer the following:

- $\|A\|$, created by the translation of A, can only contain $df(i_1,p,i_2,x)$ if $i_1 = \hat{p}$, for $p \in prem(a_2)$ $i_2 = \hat{x}$ and $p \equiv x$ for $x \in \mathcal{P}$.
- $\|A\|$ can only contain $ca(i_2,i_3)$ if $i_2 = \hat{x}$, $i_3 = \hat{c_1}$, $c_1 = concl(a_1)$ and $c_1 \not\equiv x$

Thus, we infer that there is an undercut relation r between a_1 and a_2 (case b). We note that $g = \|r\|$, and therefore, $g \in \|I_A(undercut)\|$. Thus (2) holds.

c) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_3 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1,val_1,i_2,val_2),\ ca(i_2,i_3)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$, $ca(i_1,i_2)$ a *conflict application* between two I-nodes i_1,i_2 and $df(i_1,val_1,i_2,val_2)$ a *default rephrase* between the I-nodes i_2,i_3. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1,a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern is $\langle\langle cp_1 \rangle\rangle.w_i$ and that one of the two variables of the conflict pattern is the $\langle\langle premp_2 \rangle\rangle.w_i$, and that there is a join between them, we infer the following:

- $\|A\|$, created by the translation of A, can only contain $df(i_1,c_1,i_2,x)$ if $i_1 = \hat{c_1}$, $c_1 = concl(a_1)$, $i_2 = \hat{x}$ and $c_1 \equiv x$, for $c_1,x \in \mathcal{P}$.
- $\|A\|$ can only contain $ca(i_2,i_3)$ if $i_2 = \hat{x}$, , $i_3 = \hat{p}$, for $p \in prem(a_2)$ and $p \not\equiv x$

Thus, we infer that there is a rebut relation r between a_1 and a_2 (case c). We note that $g = \|r\|$, and therefore, $g \in \|I_A(rebut)\|$. Thus (2) holds.

d) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_4 \rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1,c_1,i_2,x),\ df(i_3,p,i_4,y),\ ca(i_2,i_4)\ )\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$, $df(i_1,c_1,i_2,x)$ a *default rephrase* between the I-nodes i_1,i_2, $df(i_3,p,i_4,y)$ a *default rephrase* between the I-nodes i_3,i_4 and $ca(i_2,i_4)$ a *conflict application* between the i_2,i_4. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1,a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern eqp_1 is $\langle\langle cp_1 \rangle\rangle.w_i$, that one of the two variables in the equivalence pattern eqp_2 is $\langle\langle premp_2 \rangle\rangle.w_i$ and that the conflict pattern joins with the variables $\langle\langle eqp_1 \rangle\rangle.w_{i2}$ and $\langle\langle eqp_2 \rangle\rangle.w_{i2}$, we infer the following:

- $\|A\|$, created by the translation of A, can only contain $df(i_1,c_1,i_2,x)$ if $i_1 = \hat{c_1}$, $c_1 = concl(a_1)$, $i_2 = \hat{x}$ and $c_1 \equiv x$, for $c_1,x \in \mathcal{P}$.
- $\|A\|$ can only contain $df(i_3,p,i_4,y)$ if $i_3 = \hat{p}$, for $p \in prem(a_2)$, $i_4 = \hat{y}$ and $p \equiv y$, for $y \in \mathcal{P}$.
- $\|A\|$ can only contain $ca(i_2,i_4)$ if $i_2 = x$, $i_4 = y$ and $x \not\equiv y$

Thus, we infer that there is an undercut relation r between a_1 and a_2 (case d). We note that $g = \|r\|$, and therefore, $g \in \|I_A(undercut)\|$. Thus (2) holds.

- **(*Attack*)** Regarding the *attack* relation pattern, we recall that:

  $I_A(attack) = I_A(rebut) \cup I_A(undercut)$

  The graph pattern generated by its translation, according to the rule $r_{12}.3$ is:

  $\langle\!\langle attack \rangle\!\rangle = \langle\!\langle rebut \rangle\!\rangle$ *UNION* $\langle\!\langle undercut \rangle\!\rangle$

  (1) For the left to right direction, we assume $r \in I_A(attack)$. This means that $r \in I_A(rebut)$ or $r \in I_A(undercut)$. According to the previous two cases, we will have that $\|r\| \in \mathcal{E}_{\|A\|}(rebut)$ or $\|r\| \in \mathcal{E}_{\|A\|}(undercut)$, which, according to the semantics of UNION means that $\|r\| \in \mathcal{E}_{\|A\|}(attack)$ and as a result $\|I_A(attack)\| \subseteq \mathcal{E}(\langle\!\langle attack \rangle\!\rangle, \|K_A\|)$

  (2) The opposite direction is equally trivial.

- **(*Endorse*)** By the semantics of section 3.3.3, we have that:

  $I_A(endorse) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A$ s.t. concl$(a_1) \equiv$ concl(a_2) $\}$

  Moreover, from table 4.3 (rule $(r_{12}.4)$), we have that:

  $$\langle\!\langle \textbf{endorse} \rangle\!\rangle = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \Big[(\langle\!\langle ap_1 \rangle\!\rangle.w_{ra}\ conclusion\ \langle\!\langle cp_2 \rangle\!\rangle.w_i) \ \vee$$
  $$(\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}\ conclusion\ \langle\!\langle cp_1 \rangle\!\rangle.w_i) \ \vee \ \langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i; w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i} \Big]$$

  We decompose and rewrite $\langle\!\langle endorse \rangle\!\rangle$ into smaller graph patterns and rewrite it as:

  such that $\langle\!\langle endorse \rangle\!\rangle = gp_1$ *UNION* gp_2 *UNION* gp_3

  such that $gp_1 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge (\langle\!\langle ap_1 \rangle\!\rangle.w_{ra}\ conclusion\ \langle\!\langle cp_2 \rangle\!\rangle.w_i)$

  $gp_2 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge (\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}\ conclusion\ \langle\!\langle cp_1 \rangle\!\rangle.w_i)$

  $gp_3 = \langle\!\langle ap_1 \rangle\!\rangle \wedge \langle\!\langle ap_2 \rangle\!\rangle \wedge \langle\!\langle eqp \rangle\!\rangle_{w_{i1}=\langle\!\langle cp_1 \rangle\!\rangle.w_i; w_{i2}=\langle\!\langle cp_2 \rangle\!\rangle.w_i})$

  From the SPARQL semantics, we have the following:

  - $\sigma \in \varepsilon_A(endorse) = \sigma \in \varepsilon_A(gp_1)$ *or* $\sigma \in \varepsilon_A(gp_2)$ *or* $\sigma \in \varepsilon_A(gp_3)$ *or* $\sigma \in \varepsilon_A(gp_4)$

  - $\mathcal{E}_{\|A\|}(endorse) = \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2) \cup \mathcal{E}_{\|A\|}(gp_3)$

(1) For the left to right direction we assume $r \in I_A(endorse)$. This means that $r = (a_1\ a_2)$, where $a_1, a_2 \in A$. Otherwise, it holds that either $concl(a_1) = concl(a_2)$, or $concl(a_1) \equiv concl(a_2)$.

Each of these two cases, according to the data mapping tables 4.1, is translated as follows:

a) $\|r\| = \{\|a_1\|, \|a_2\|\}$, where $concl(a_1) = concl(a_2)$.

b) $\|r\| = \{\|a_1\|, \|a_2\|, \|e\|\}$, where $e \in eq(A)$ and $e = (concl(a_1), concl(a_2))$.

Obviously, given that $\|I_A(endorse)\| = \{\ \|r_i\| \mid r_i \in I_A(r_i)\ \}$, we have that $\|r\| \in \|I_A(endorse)\|$ and that $\|r\| \subseteq \|A\|$. Furthermore, by the proof of an argument pattern above and given that ap_1, ap_2 are random argument patterns which match any argument, we have that $\|a_1\| \in \mathcal{E}_{\|A\|}(\langle\langle ap_1\rangle\rangle)$ and $\|a_2\| \in \mathcal{E}_{\|A\|}(\langle\langle ap_2\rangle\rangle)$

a) If r belongs to the case (a), then for $\sigma_1 = \{(\langle\langle ap_1\rangle\rangle.w_{ra},\ u(\hat{a_1})), (\langle\langle ap_2\rangle\rangle.w_{ra}, u(\hat{a_2})), (\langle\langle cp_2\rangle\rangle.w_i,\ u(\hat{c_1}))\ \}$, where $c_1 = concl(a_1)$, we get that $\sigma_1 \in \varepsilon_{\|A\|}(gp_1)$, while for $\sigma_2 = \{(\langle\langle ap_1\rangle\rangle.w_{ra},\ u(\hat{a_1})), (\langle\langle ap_2\rangle\rangle.w_{ra},\ u(\hat{a_2})), (\langle\langle cp_1\rangle\rangle.w_i,\ u(\hat{c_2}))\ \}$ where $c_2 = concl(a_2)$, we get that $\sigma_2 \in \varepsilon_{\|A\|}(gp_2)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\langle endorse\rangle\rangle)$. Thus (1) holds.

b) If r belongs to the case (b), then for

$\sigma = \{(\langle\langle ap_1\rangle\rangle.w_{ra},\ u(\hat{a_1})), (\langle\langle ap_2\rangle\rangle.w_{ra}, u(\hat{a_2})), (\langle\langle eqp\rangle\rangle.w_{i1}, u(\hat{c_1}))\ , (\langle\langle eqp\rangle\rangle.w_{i2}, u(\hat{c_2}))\ , (\langle\langle eqp\rangle\rangle.w_{ma},\ u(\hat{e}))\ \}$

where $c_1 = concl(a_1)$ and $c_2 = concl(a_2)$ we get that $\sigma \in \varepsilon_{\|A\|}(gp_3)$, which generates a $df(\hat{c_1}, c_1, \hat{c_2}, c_2)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_3)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\langle endorse\rangle\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle endorse\rangle\rangle)$. By the definition of $\langle\langle endorse\rangle\rangle$ we have one of the four cases:

a) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_1\rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ (u(ra_1)\ conclusion\ u(i))\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1\rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2\rangle\rangle)$, $u(ra_1)$ is the uri of an RA-node instance, and $u(i)$ is the uri of an I-node instance. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. Moreover, it will also hold that $u(\hat{a_1}) = u(ra_1)$ and $u(\hat{c_2}) = u(i)$, for $c_2 = concl(a_2)$. As a result given that there is the triple $(u(\hat{a_1})\ conclusion\ u(c_2))$ we get that $concl(a_1) = concl(a_2)$, which means that there is an endorse relation between a_1, a_2 (case a). We note that $g = \|r\|$, and therefore, $g \in \|I_A(endorse)\|$. Thus (2) holds.

b) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_2\rangle\rangle)$, then the proof goes accordingly:

c) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_3 \rangle\rangle)$, then g will be of the form:

$g = \{\, g_1, g_2, df(i_1, c_1, i_2, c_2)\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1 \rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2 \rangle\rangle)$ and $df(i_1, c_1, i_2, c_2)$ a *default rephrase* between the I-nodes i_2, i_3. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern is $\langle\langle cp_1 \rangle\rangle.w_i$ and the other is the $\langle\langle cp_2 \rangle\rangle.w_i$ we get that $\|A\|$, created by the translation of A, can only contain $df(i_1, c_1, i_2, c_2)$ if $i_1 = \hat{c_1}$, $c_1 = concl(a_1)$, $i_2 = \hat{c_2}$, $c_2 = concl(a_2)$ and $c_1 \equiv c_2$, for $c_1, c_2 \in \mathcal{P}$.

As a result, we infer that there is an endorse relation r between a_1 and a_2 (case b). We note that $g = \|r\|$, and therefore, $g \in \|I_A(endorse)\|$. Thus (2) holds.

- **(*Back*)** By the semantics of section 3.3.3, we have that:

$I_A(back) = \{P_{a_1 \to a_2} = (a_1\ a_2) \mid a_1, a_2 \in A$ s.t. concl(a_1) $\equiv p_i$ for some $p_i \in$ prem(a_2) $\}$

Moreover, from table 4.3 (rule $(r_{12}.5)$), we have that:

$$\langle\langle \textbf{back} \rangle\rangle = \langle\langle ap_1 \rangle\rangle \wedge \langle\langle ap_2 \rangle\rangle \wedge \Big((\langle\langle ap_2 \rangle\rangle.w_{ra}\ premise\ \langle\langle cp_1 \rangle\rangle.w_i) \quad \bigvee$$
$$\langle\langle eqp \rangle\rangle_{w_{i1} = \langle\langle cp_1 \rangle\rangle.w_i} \wedge (\langle\langle ap_2 \rangle\rangle.w_{ra}\ premise\ \langle\langle eqp \rangle\rangle.w_{i2}) \Big)$$

We decompose and rewrite $\langle\langle back \rangle\rangle$ into smaller graph patterns and rewrite it as:

$\langle\langle back \rangle\rangle = gp_1\ UNION\ gp_2$

such that

$gp_1 = \langle\langle ap_1 \rangle\rangle \wedge \langle\langle ap_2 \rangle\rangle \wedge (\langle\langle ap_2 \rangle\rangle.w_{ra}\ premise\ \langle\langle cp_1 \rangle\rangle.w_i)$

$gp_2 = \langle\langle ap_1 \rangle\rangle \wedge \langle\langle ap_2 \rangle\rangle \wedge \langle\langle eqp \rangle\rangle_{w_{i1} = \langle\langle cp_1 \rangle\rangle.w_i} \wedge (\langle\langle ap_2 \rangle\rangle.w_{ra}\ premise\ \langle\langle eqp \rangle\rangle.w_{i2})$

From the SPARQL semantics, we have the following:

- $\sigma \in \varepsilon_A(back) = \sigma \in \varepsilon_A(gp_1)\ or\ \sigma \in \varepsilon_A(gp_2)$

- $\mathcal{E}_{\|A\|}(back) = \mathcal{E}_{\|A\|}(gp_1) \cup \mathcal{E}_{\|A\|}(gp_2)$

(1) For the left to right direction, we assume $r \in I_A(back)$. This means that
$r = P_{a_1 \to a_2} = (a_1\ a_2)$, where $a_1, a_2 \in A$. More precisely, there is a *back* relation between a_1 and a_2 in one of the following two cases:

a) $concl(a_1) \in prem(a_2)$
b) $concl(a_1) \equiv p$, for $p \in prem(a_2)$

According to the data mapping of table 4.1, for each of these cases, we have that:

a) $\|r\| = \{\|a_1\|, \|a_2\|\}$ and for $c_1 = concl(a_1)$ it holds that $(u(\hat{c_1})$ *premise* $u(\hat{a_2})) \in \|a_2\|$.

b) $\|r\| = \{\|a_1\|, \|a_2\|, \|e\|\}$, where $e \in eq(A)$ with $e = (concl(a_1), p)$.

Obviously, given that $\|I_A(back)\| = \{\ \|r_i\| \mid r_i \in I_A(r_i)\ \}$, we have that $\|r\| \in \|I_A(back)\|$ and that $\|r\| \subseteq \|A\|$. Furthermore, by the proof of an argument pattern above and given that ap_1, ap_2 are random argument patterns which match any argument, we have that $\|a_1\| \in \mathcal{E}_{\|A\|}(\langle\langle ap_1\rangle\rangle)$ and $\|a_2\| \in \mathcal{E}_{\|A\|}(\langle\langle ap_2\rangle\rangle)$

a) If r belongs to the case (a), then for $\sigma = \{(\langle\langle ap_1\rangle\rangle.w_{ra},\ u(\hat{a_1})), (\langle\langle ap_2\rangle\rangle.w_{ra}, u(\hat{a_2})), (\langle\langle cp_1\rangle\rangle.w_i,$ $u(\hat{c_1}))\ \}$, where $c_1 = concl(a_1)$, we get that $\sigma \in \varepsilon_{\|A\|}(gp_1)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_1)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\langle back\rangle\rangle)$. Thus (1) holds.

b) If r belongs to the case (b), then for

$\sigma = \{(\langle\langle ap_1\rangle\rangle.w_{ra},\ u(\hat{a_1})), (\langle\langle ap_2\rangle\rangle.w_{ra}, u(\hat{a_2})), (\langle\langle eqp\rangle\rangle.w_{i1}, u(\hat{c_1}))\ , (\langle\langle eqp\rangle\rangle.w_{i2}, u(\hat{p}))\ , (\langle\langle eqp\rangle\rangle.w_{ma},$ $u(\hat{e}))\ \}$

where $c_1 = concl(a_1)$ and $p \in prem(a_2)$ we get that $\sigma \in \varepsilon_{\|A\|}(gp_2)$, which generates a $df(\hat{c_1}, c_1, \hat{p}, p)$ and therefore, by the definition of $\mathcal{E}_{\mathcal{G}}$: $\|r\| \in \mathcal{E}_{\|A\|}(gp_2)$ and $\|r\| \in \mathcal{E}_{\|A\|}(\langle\langle back\rangle\rangle)$. Thus (1) holds.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle back\rangle\rangle)$. By the definition of $\langle\langle back\rangle\rangle$ we have one of the two cases:

a) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_1\rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ (u(ra_2)$ *premise* $u(i))\}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1\rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2\rangle\rangle)$, $u(ra_2)$ is the uri of an RA-node instance, and $u(i)$ is the uri of an I-node instance. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and $g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. Moreover, it will also hold that $u(\hat{a_2}) = u(ra_2)$ and $u(\hat{p}) = u(i)$, for $p \in prem(a_2)$. As a result given that there is the triple $(u(\hat{a_2})$ *premise* $u(\hat{p}))$ we get that $concl(a_1) = p$, which means that there is a back relation between a_1, a_2 (case b). We note that $g = \|r\|$, and therefore, $g \in \|I_A(back)\|$. Thus (2) holds.

b) If $g \in \mathcal{E}_{\|A\|}(\langle\langle gp_2\rangle\rangle)$, then g will be of the form:

$g = \{\ g_1,\ g_2,\ df(i_1, c_1, i_2, p),\ (ra_2$ *premise* $u(i_2))\ \}$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle ap_1\rangle\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle ap_2\rangle\rangle)$, $u(ra_2)$ is the uri of an RA-node instance, and $u(i_2)$ is the uri of an I-node instance and $df(i_1, c_1, i_2, p)$ a *default rephrase* between the I-nodes i_2, i_3. We showed from the previous case (argument pattern) that $g_1 \in \|I_A(ap_1)\|$ with $g_1 = \|a_1\|$ and

$g_2 \in \|I_A(ap_2)\|$ with $g_2 = \|a_2\|$, where $a_1, a_2 \in A$. In addition, given that one of the two variables in the equivalence pattern is $\langle\!\langle cp_1 \rangle\!\rangle.w_i$ and that $(ra_2\ premise\ u(i_2)) \in G$ we get that $\|A\|$, created by the translation of A, can only contain $df(i_1, c_1, i_2, p)$ if $i_1 = \hat{c_1}$, $c_1 = concl(a_1)$, $i_2 = \hat{p}$ and $c_1 \equiv p$ and can only contain the triple (ra_2 premise $u(i_2)$), if $p \in prem(a_2)$.

As a result, we infer that there is a back relation r between a_1 and a_2 (case b). We note that $g = \|r\|$, and therefore, $g \in \|I_A(back)\|$. Thus (2) holds.

- **(*Support*)** Regarding the *support* relation pattern, we recall that:

  $I_A(support) = I_A(endorse) \cup I_A(back)$

  The graph pattern generated by its translation, according to the rule $r_{12}.6$ is:

  $\langle\!\langle \boldsymbol{support} \rangle\!\rangle = \langle\!\langle endorse \rangle\!\rangle\ UNION\ \langle\!\langle back \rangle\!\rangle$

  (1) For the left to right direction, we assume $r \in I_A(support)$. This means that $r \in I_A(endorse)$ or $r \in I_A(back)$. According to the previous two cases, we will have that $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle endorse \rangle\!\rangle)$ or $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle back \rangle\!\rangle)$, which, according to the semantics of UNION means that $\|r\| \in \mathcal{E}_{\|A\|}(\langle\!\langle support \rangle\!\rangle)$ and as a result $\|I_A(support)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle support \rangle\!\rangle)$

  (2) The opposite direction is equally trivial.

Path pattern

We assume a path pattern, *pp*. We have to show that for all its different types, it holds that $\|I_A(pp)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle pp \rangle\!\rangle)$(1) and $\mathcal{E}_{\|A\|}(\langle\!\langle pp \rangle\!\rangle) \subseteq \|I_A(pp)\|$(2). As a result we have the following cases:

- If *pp* is a simple relation pattern *relp*, then the proof was given before.

- If *pp* is of the form pp_1/pp_2, with pp_1, pp_2 path patterns, we recall that:

  $I_A(pp_1/pp_2) = \{P_{a_1 \to a_n} = P_{a_1 \to a_m} \cdot P_{a_m \to a_n} \mid P_{a_1 \to a_m} \in I_A(pp_1) \text{ and } P_{a_m \to a_n} \in I_A(pp_2)\}$

  The graph pattern generated by its translation according to the rule ($r_{13}.2$) is:

  $\langle\!\langle pp_1/pp_2 \rangle\!\rangle = \langle\!\langle pp_1 \rangle\!\rangle \wedge \langle\!\langle pp_2 \rangle\!\rangle_{\langle\!\langle ap_1 \rangle\!\rangle.w_{ra} = \langle\!\langle pp_1 \rangle\!\rangle.\langle\!\langle ap_2 \rangle\!\rangle.w_{ra}}$

  (1) For the left to right direction, we assume $p \in I_A(pp)$. Given that pp lies between two random argument patterns ap_1 and ap_n and according to the definition 20, we have that:

  $p = (a_1\ \dots\ a_j\ \dots\ a_n)$, for which $p_1 = (a_1 \dots\ a_j) \in I_A(pp_1)$ and
  $p_2 = (a_j\ \dots\ a_n) \in I_A(pp_2)$.

The translation of p into RDF is $\|p\| = \{\|a_1\| \ \dots \ \|a_j\|, \dots, \|a_n\|\}$ and we need to show that $\|p_1\| = \{\|a_1\|, \dots \|a_j\|\} \in \mathcal{E}_{\|A\|}(\langle\langle pp_1 \rangle\rangle)$ and $\|p_2\| = \{\|r_j\|, \dots, \|r_n\|\} \in \mathcal{E}_{\|A\|}(\langle\langle pp_2 \rangle\rangle)$. The proof proceeds depending on the type of pp_1 and pp_2. If any of them is a relation pattern, then the proof is reduced to the previous case. If it is of the current type, the proof proceeds in a recursive way until it reaches the relation pattern case, while if it is of any of the following types, they are shown bellow.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle pp_1/pp_2 \rangle\rangle)$. By the definition of $\langle\langle pp_1/pp_2 \rangle\rangle$ we have that g will be of the form:

$g = g_1, g_2$

where $g_1 \in \mathcal{E}_{\|A\|}(\langle\langle pp_1 \rangle\rangle)$ and $g_2 \in \mathcal{E}_{\|A\|}(\langle\langle pp_2 \rangle\rangle)$. Based on the particular type of pp_1 and pp_2, the proof proceeds respectively.

- If pp is of the form $pp * n$, for $n > 1$, we have:
  - if $n = 1$, then $I_A(pp * 1) = I_A(pp)$
  - if $n > 1$, then $I_A(pp * n) = I_A(pp/pp * (n-1))$

  The graph pattern generated by its translation according to the rule $(r_{13}.3)$ is:

  - if $n = 1$, then $\langle\langle pp{*}n \rangle\rangle = \langle\langle pp \rangle\rangle$
  - if $n > 1$, then $\langle\langle pp{*}n \rangle\rangle = \langle\langle pp/pp{*}(n\text{-}1) \rangle\rangle$

(1) For the left to right direction, we have:

- if $n = 1$, then we have to show that if $p \in I_A(pp * 1)$, then $\|p\| \in \mathcal{E}_{\|A\|}(\langle\langle pp \rangle\rangle)$, which also depends on the particular form of pp.

-if $n > 1$, then we have to show that if $p \in I_A(pp * n)$, then $\|p\| \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}n \rangle\rangle)$. But since $pp * n = pp/pp * (n-1)$, this case is reduced to the previous case ($pp = pp_1/pp_2$), where $pp_1 = pp$ and $pp_2 = pp * (n-1)$.

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}n \rangle\rangle)$. By definition of $\langle\langle pp{*}n \rangle\rangle$ we have:

- if $n = 1$, then $\langle\langle pp{*}n \rangle\rangle = \langle\langle pp \rangle\rangle$, and the continuity depends on the type of pp.

- if $n > 1$, then $\langle\langle pp/pp{*}(n\text{-}1) \rangle\rangle$ and the case is reduced to the previous case ($pp = pp_1/pp_2$), where $pp_1 = pp$ and $pp_2 = pp * (n-1)$.

- If pp is of the form $pp+n$, we have:

  $I_A((t)+n) = \bigcup_{k=1}^{n} I_A((t)*k)$

  The graph pattern generated by the translation of this path pattern, according to the rule $(r_{13}.4)$ is:

  $\langle\langle pp+n\rangle\rangle = \bigcup_{k=1}^{n}\langle\langle pp{*}k\rangle\rangle = \langle\langle pp*1\rangle\rangle$ *UNION .. UNION* $\langle\langle pp*n\rangle\rangle$

  (1) For the left to right direction, we assume $p \in I_A(pp+n)$. This is translated as $p \in I_A(t*1)$ or $p \in I_A(t*2)$ .. or $p \in I_A(t*n)$. We have to prove that $\|p\| \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}1\rangle\rangle)$ or $\|p\| \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}2\rangle\rangle)$ .. or $\|p\| \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}n\rangle\rangle)$. According to the semantics of UNION in SPARQL, this means that $\|p\| \in \mathcal{E}_{\|A\|}(\langle\langle pp{+}n\rangle\rangle)$ and that $\|I_A(pp+n)\| \subseteq \mathcal{E}_{\|A\|}(\langle\langle pp{+}n\rangle\rangle)$. Each of these n cases are reduced to the previous type of path pattern ($pp*n$).

  (2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\langle pp{+}n\rangle\rangle)$. By definition of $\langle\langle pp{+}n\rangle\rangle$, this means that it will for g will hold one of the following:

  $g \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}1\rangle\rangle)$, or

  $g \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}2\rangle\rangle)$, or

  ...

  $g \in \mathcal{E}_{\|A\|}(\langle\langle pp{*}n\rangle\rangle)$

  Depending on which of the n cases holds, the proof is reduced to the previous case where $pp = pp*n$.

Summing up, we see that all cases result to a sequence of relation patterns, the proof of which was given in the first case.

Dialogue pattern

We assume a path pattern, *dp*. We have to show that for both of its different forms, it holds that $\|I_A(dp)\| \subseteq \mathcal{E}_{\|A\|}(\langle\langle dp\rangle\rangle)$ (1) and $\mathcal{E}_{\|A\|}(\langle\langle dp\rangle\rangle) \subseteq \|I_A(dp)\|$(2).

- If dp an argument pattern *ap*, then: $I_A(t) = \{P_{a\to a} = a \mid a \in I_A(ap)\}$

  The graph pattern generated by its translation according to the rule $(r_{14}.1)$ is $\langle\langle dp\rangle\rangle = \langle\langle ap\rangle\rangle$ and the proof for both directions is the same to the case of argument pattern.

- If $dp = ap\ pp\ dp'$, where ap an argument pattern, pp a path pattern and dp' another dialogue pattern, we have:

$I_A(t) = \{P_{a_0 \to a_n} = (a_0\ a_1\ \dots\ a_x \dots\ a_n) \mid a_0 \in I_A(ap),\ P_{a_0 \to a_x} \in I_A(pp),\ a_x \in I_A(head(dp'))$ and $P_{a_x \to a_n} \in I_A(dp')\}$

The graph pattern generated by its translation according to the rule $(r_{14}.2)$ is:

$\langle\!\langle dp \rangle\!\rangle = \langle\!\langle ap \rangle\!\rangle \wedge \langle\!\langle pp \rangle\!\rangle_{\langle\!\langle ap_1 \rangle\!\rangle.w_{ra} = \langle\!\langle ap_0 \rangle\!\rangle.w_{ra}\ ;\ \langle\!\langle ap_2 \rangle\!\rangle.w_{ra} = \langle\!\langle head(dp') \rangle\!\rangle.w_{ra}} \wedge \langle\!\langle dp' \rangle\!\rangle$

(1) For the left to right direction, we assume $d \in I_A(dp)$. This means that:

$d = (a_1\ \dots\ a_x \dots\ a_n)$. The RDF representation of d is:

$\|d\| = \{\|a_1\|, \dots \|a_x\|, \dots \|a_n\|\}$,

For this sequence we have the following:

- $a_1 \in I_A(ap)$, and we showed that $\|a_1\| \in \mathcal{E}_{\|A\|}(\langle\!\langle ap \rangle\!\rangle)$
- $p_1 = (a_1\ \dots\ a_x) \in I_A(pp)$ and we showed that $\|p_1\| = \{\|a_1\|, \dots \|a_x\|\} \in \mathcal{E}_{\|A\|}(\langle\!\langle pp \rangle\!\rangle)$
- $a_x \in I_A(head(dp'))$ and we showed that $\|a_x\| \in \mathcal{E}_{\|A\|}(\langle\!\langle head(dp') \rangle\!\rangle)$ and finally
- $p_2 = (a_x\ \dots\ a_n) \in I_A(dp')$ and the proof for $\|p_2\| = \{\|a_x\|, \dots \|a_n\|\} \in \mathcal{E}_{\|A\|}(\langle\!\langle dp' \rangle\!\rangle)$ proceed recursively, depending on the type of dp'.

From the above cases, we infer that $\|d\| \in \mathcal{E}_{\|A\|}(\langle\!\langle dp \rangle\!\rangle)$ and thus $\|I_A(dp)\| \subseteq \mathcal{E}_{\|A\|}(\langle\!\langle dp \rangle\!\rangle)$

(2) For the opposite direction, take a set of triples $g \in \mathcal{E}_{\|A\|}(\langle\!\langle dp \rangle\!\rangle)$. By definition of $\langle\!\langle dp \rangle\!\rangle$, it means that g will have the following form:

$g = g_1, g_2, g_3$

such that $g_1 \in \mathcal{E}_{\|A\|}(\langle\!\langle ap \rangle\!\rangle)$, $g_2 \in \mathcal{E}_{\|A\|}(\langle\!\langle pp \rangle\!\rangle)$ and $g_3 \in \mathcal{E}_{\|A\|}(\langle\!\langle dp' \rangle\!\rangle)$. It holds that:

- By the case of argument pattern we have that g_1 has been created by an argument $a \in A$, for which $a \in I_A(()ap)$
- By the case of proposition pattern, we have that g_2 corresponds to a path $p = (a_1 \dots\ a_x)$, for which $p \in I_A(pp)$ and $a_x \in I_A(head(dp'))$
- We need to show that g_3 corresponds to a path $p' = (a_x\ dots\ a_n)$ such that $p' \in I_A(dp')$. To prove this, the process proceeds recursively, depending on the type of dp'.

As a result, we infer that $g \in \|I_A(dp)\|$ and thus $\mathcal{E}_{\|A\|}(\langle\!\langle ap \rangle\!\rangle) \in \|I_A(dp)\|$

□

Appendix A
Publications

Publications

The research activity related to this thesis has so far produced the following publications (ordered by publication date)

(1) Dimitra Zografistou, Giorgos Flouris, and Dimitris Plexousakis. *Argql: A declarative language for querying argumentative dialogues.* In International Joint Conference on Rules and Reasoning (RuleML), pages 230-237. Springer, 2017. **(Best Doctoral Consortium Paper)**

(2) Dimitra Zografistou, Giorgos Flouris, Theodore Patkos, and Dimitris Plexousakis. *Implementing the argql query language.* In Computational Models of Argument - Proceedings of COMMA 2018, Warsaw, Poland, 12-14 September 2018, pages 241-248, 2018.

(3) Dimitra Zografistou, Giorgos Flouris, Theodore Patkos, and Dimitris Plexousakis. *A language for graphs of interlinked arguments.* ERCIM News, 2019 (118), 2019.

(4) Towards submitting the complete work to the Journal *Argument and Computation*

Appendix B
Acronyms

AIF Argument Interchange Format

ArgQL Argumentation Query Language

AML Argumentation Markup Language

ANTLR ANother Tool for Language Recognition

AST Abstract Syntax Trees

BDI Beliefs Desires Intentions

EBNF Extended Backus-Naur Form

IBIS Issue-Based Information Systems

NLP Natural Language Processing

QID Query Intermediate Data

RDF Resource Description Framework

SPARQL Protocol and RDF Query Language